Diplomarbeit

im Studiengang Medieninformatik
an der Hochschule der Medien

Konzeption und Realisierung einer Anwendungserweiterung zur Online-Lokalisierung von Sprachressourcen

Vorgelegt von
Jakob Meister
20. April 2005

Erstprüfer: Prof. Walter Kriha
Zweitprüfer: Dipl.- Ing. Markus Leutner

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Stuttgart, 20. April 2005

Jakob Meister

Kurzfassung

Diese Diplomarbeit beschäftigt sich mit der Internationalisierung und Lokalisierung von Software. Das Ziel dieser Arbeit war die Realisierung einer Software zur Unterstützung des Übersetzungsprozesses von Sprachressourcen für die IUCCA Anwendung.

IUCCA ist ein komplexes webbasiertes Auftragsabwicklungssystem, das europaweit eingesetzt wird. Aufgrund der kontinuierlichen Weiterentwicklung von IUCCA unterliegen auch die Sprachressourcen dieses Systems einem fortlaufenden Übersetzungsprozess.

Dieser Prozess erwies sich allerdings als aufwändig und fehleranfällig. Daher wurde im Rahmen dieser Arbeit ein neuer Prozess definiert und eine auf dem Open-Source Framework Struts basierende Anwendung realisiert. Diese Anwendung bildet den neuen Übersetzungsprozess ab und bindet diesen in das *eXtreme Programming*-Vorgehensmodell der IUCCA-Entwicklung ein.

Neben der Analyse der Prozesse wird auch die Realisierung der Anwendung beschrieben. Darüber hinaus wird die bei der Internationalisierung auftretende Zeichensatzproblematik untersucht und der Unicode-Zeichensatz vorgestellt. Weiterhin wird gezeigt, wie die Programmiersprache Java die Internationalisierung von Anwendungen unterstützt.

Abstract

The concern of this diploma thesis is the internationalization and localization of software. The goal of this thesis was the realization of an application, which supports the translation process of language resources for the IUCCA application.

IUCCA is a complex web based system for order processing, which is used throughout Europe. Due to the continuous development of IUCCA also the languages resources are subject to a continual translation process.

This process proved to be costly and error-prone. In the scope of this thesis a new process was defined and an application was implemented, based on the open source framework Struts. This application represents the new process and integrates it into the eXtreme Programming approach of the IUCCA development.

Along with the analysis of the processes also the realization of this application will be described. Furthermore the problem of conflicting character sets is shown und the Unicode character set will be introduced. Eventually also the support of internationalization through the Java programming language will be explained.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ASCII	American Standard Code for Information Interchange
BMP	Basic Multilingual Plane
CJK	Chinesisch Japanisch Koreanisch
CSV	Comma Separated Value
CVS	Concurrent Versions System
DBCP	Database Connection Pool
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICE	International Electrotechnical Commision
ISO	International Organization for Standardization
IUCCA	Internet based Used Car Center Application
J2SE	Java 2 Standard Edition
J2EE	Java 2 Enterprise Edition
JAR	Java Archiv
JDK	Java Development Kit
JNDI	Java Naming And Directory Interface.
JSP	Java Server Pages
OSE	Open Servlet Environment
SIP	Supplementary Ideographic Plane
SMP	Supplementary Multilingual Plane
TROIA	Text Resources Online Integration and Administration
UML	Unified Modeling Language
UTF	Unicode Transformation Format
WAR	Web Archiv
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformation

1 Überblick

Durch die Globalisierung und die weltweite Verbreitung des Internets muss Software immer häufiger in mehreren Ländern und Sprachen einsetzbar sein. Diese Anforderung stellt sich vor allem für Software, die für international agierende Unternehmen erstellt wird. Die Herausforderungen sind dabei zum einen, die Software so zu entwickeln, dass sie sich für den länderübergreifenden Einsatz eignet. Andererseits müssen auch effektive Prozesse für die Anpassung der länderabhängigen Teile einer Software aufgesetzt werden.

In dieser Diplomarbeit werden sowohl die technischen Grundlagen für die Internationalisierung von Anwendungen als auch die Prozesse für die Anpassung deren länderabhängigen Komponenten betrachtet. Im Mittelpunkt steht dabei der Übersetzungsprozess für eine komplexe, webbasierte Anwendung, die von der T-Systems International GmbH für die DaimlerChrysler AG entwickelt wird. Für die Optimierung dieses Prozesses wurde im Rahmen dieser Arbeit eine Software entworfen und realisiert.

Diese Arbeit ist folgendermaßen aufgebaut:

Kapitel 2: Hier wird die T-Systems International GmbH vorgestellt und das »IUCCA« Projekt beschrieben.

Kapitel 3: Dieses Kapitel zeigt die Problemstellung auf und definiert die Zielsetzung dieser Arbeit.

Kapitel 4: Bevor die Lösung der Problemstellung beschrieben wird, werden in diesem Kapitel die Grundlagen für die Internationalisierung von Software besprochen. Dabei wird im Zuge der Zeichensatzproblematik auf Unicode eingegangen, weiterhin wird gezeigt wie die Programmiersprache Java die Internationalisierung von Software unterstützt.

Kapitel 5: In diesem Kapitel wird die IUCCA-Anwendung beschrieben. Es wird dargestellt, wie die Mehrländerfähigkeit in IUCCA realisiert wurde. Anschließend wird gezeigt welche Probleme aus dem aktuellen Übersetzungsprozess in Bezug auf den Entwicklungsprozess von IUCCA entstehen.

Kapitel 6: Dieses Kapitel erläutert das Vorgehen bei der Analyse und Planung der Software TROIA , die für die Optimierung des Übersetzungsprozesses entwickelt wurde.

Kapitel 7: Hier werden die Technologien vorgestellt, die bei der Implementierung von TROIA eingesetzt wurden. Dabei liegt der Schwerpunkt bei dem Apache Struts Framework, auf dem die Anwendung TROIA aufbaut.

Kapitel 8: Dieses Kapitel widmet sich der Verwirklichung von TROIA. Zu Beginn werden die Architektur und die darin verwendeten Software Entwurfsmuster beschrieben. Anschließend wird die Funktionsweise von TROIA an Hand von *Screenshots* dargelegt.

2 Einführung

Die vorliegende Diplomarbeit entstand in Zusammenarbeit mit dem Team »Remarketing Solutions« der T-Systems International GmbH. In diesem Kapitel wird ein kurzer Überblick über die T-Systems International GmbH gegeben. Anschließend wird das vom Team »Remarketing Solutions« entwickelte Projekt »IUCCA«, aus dem die Problemstellung dieser Diplomarbeit hervorgegangen ist, vorgestellt.

2.1 Das Unternehmen

T-Systems International GmbH ist ein Tochterunternehmen der Deutschen Telekom AG. Die T-Systems GmbH entstand im Jahr 2001 durch die Fusionierung mehrerer Telekom Tochterunternehmen mit dem debis Systemhaus, einer ehemaligen Tochter der DaimlerChrysler AG.

T-Systems ist einer der führenden Dienstleister für Informations- und Kommunikationstechnik in Europa. Im Konzern Deutsche Telekom steht die Marke T-Systems für das strategische Geschäftsfeld »Geschäftskunden«: Dies umfasst sowohl rund 60 multinational agierende Konzerne und große Institutionen der öffentlichen Hand als auch ca. 160.000 große und mittelständische Unternehmen.

Die T-Systems bietet folgende Dienstleistungen rund um Informations- und Kommunikationstechnik (engl. kurz: ICT) an.

- *ICT Infrastructure Management* – Prozessneutrale Infrastrukturleistungen.
- *Business Solution Design & Implementation* – Entwicklung, Implementierung und Betrieb von Lösungen zur Geschäftsunterstützung.
- *Business Process Management* – Betrieb kompletter Geschäftsprozesse.

2.2 Das »IUCCA« Projekt

Das Akronym IUCCA steht für *Internet based Used Car Center Application.* IUCCA ist ein webbasiertes Auftragsabwicklungs- und Management-Informationssystem, das von dem »Remarketing Solutions«-Team der T-Systems für die TruckStores[1] der DaimlerChrysler AG entwickelt wird.

»TruckStore« ist die neue Bezeichnung der Mercedes-Benz Nutzfahrzeug-Gebrauchtwagen-Center. In diesen Centern werden gebrauchte Nutzfahrzeuge und Dienstleitungen im Umfeld dieser Nutzfahrzeuge angeboten. Mit dem neuen Namen wurde auch ein neues Konzept für diese Center geschaffen. In erster Linie soll dieses

[1] Siehe auch www.truckstore.com

Konzept eine durchgängige Professionalität und Kundenorientierung an allen Standorten in Europa gewährleisten. Erreicht werden soll das durch ein einheitliches Corporate Design und europaweit standardisierte Geschäftsprozesse.

Daher wurde mit IUCCA ein Auftragsabwicklungssystem geschaffen, welches die Geschäftsprozesse der TruckStores abbildet und diese über ein zentrales System europaweit zur Verfügung stellt.

In Abbildung 1 sind die einzelnen Geschäftsprozesse, vom Ankauf über den Verkauf bis hin zur Nachkalkulation eines gebrauchten LKWs, dargestellt.

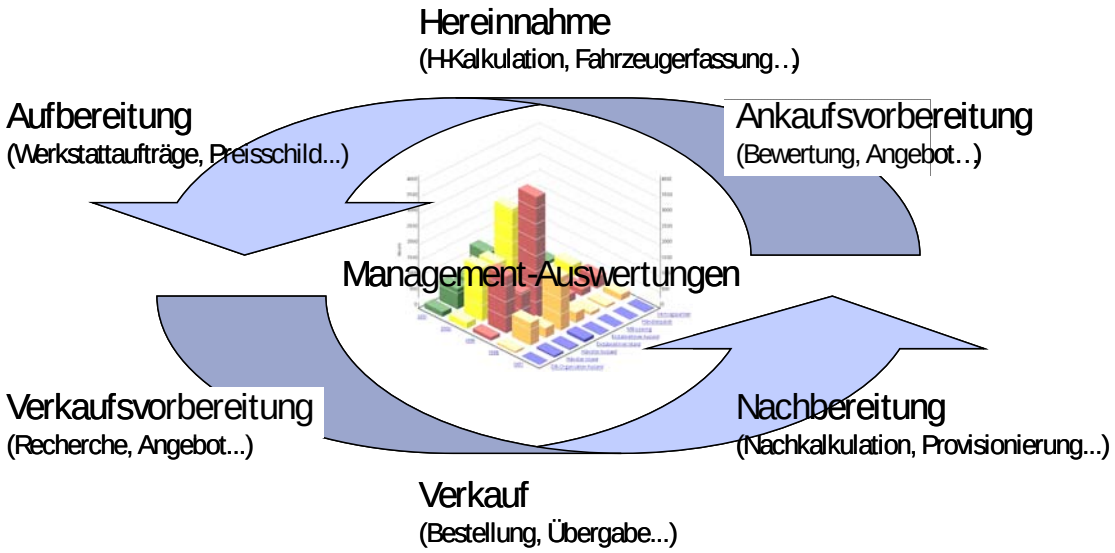

Abbildung 1: Geschäftsprozesse in IUCCA

Ein Modul von IUCCA ist das Management-Informationssystem IUCCA-MIS. Dieses Informationssystem bietet geschäftsrelevante Auswertungen unterschiedlicher Granularität über die Daten aller TruckStores in Europa.

Das IUCCA System ist also eine komplexe Anwendung mit vielen unterschiedlichen Benutzeroberflächen, Modulen und Schnittstellen zu anderen Systemen. Da das IUCCA-System europaweit eingesetzt wird, muss es internationalisierbar sein. Das heißt es muss möglich sein, den jeweiligen Ländern unterschiedliche Module, z.B. zum Berechnen von Provisionen oder Steuern, zur Verfügung zu stellen. Weiterhin soll es möglich sein, IUCCA über unterschiedliche Schnittstellen an die bestehenden Systeme der einzelnen Länder zu koppeln. Schließlich müssen auch die internetbasierten Benutzeroberflächen für jedes Land in der entsprechenden Landessprache darstellbar sein.

3 Problemstellung und Zielsetzung

Nachdem im letzten Kapitel das IUCCA- System vorgestellt wurde, wird in diesem Kapitel die Problemstellung, die sich aus der Internationalisierung von IUCCA ergibt, erläutert und die Zielsetzung dieser Diplomarbeit definiert.

3.1 Problemstellung

Wie in Kapitel 2.2 bereits erwähnt ist IUCCA für den europaweiten Einsatz konzipiert. Momentan wird IUCCA schon in Deutschland, Holland, Frankreich, Spanien, Portugal und Italien betrieben. Weiterhin kommen in der nächsten Zeit Belgien, Griechenland und Russland dazu. Parallel zu der Einführung in den einzelnen Ländern wird das IUCCA-System ständig weiterentwickelt und erweitert. Dabei können die Erweiterungen entweder nur ein bestimmtes Land betreffen, z.B. in Form einer Anpassung an nationales Steuerrecht, oder auch Auswirkungen auf alle Länder haben.

Die Einführung von IUCCA in neue Länder sowie die kontinuierliche Weiterentwicklung haben zur Folge, dass die Textressourcen von IUCCA einem andauernden Übersetzungsprozess unterliegen. Für jedes zusätzliche Land müssen die gesamten Texte übersetzt werden. Da durch die Erweiterungen neue Texte hinzukommen, müssen auch die Textressourcen für bereits bestehende Länder gepflegt und aktualisiert werden. Des weiteren ist zu beachten, dass in den einzelnen Ländern unterschiedliche Versionen von IUCCA eingesetzt werden.

Mit der zunehmenden Anzahl der zu pflegenden Sprachen zeigt sich, dass der Prozess der Übersetzung, so wie er momentan im IUCCA-Projekt gehandhabt wird, nicht effektiv genug ist, um mit der raschen Entwicklung des Projektes Schritt halten zu können. Zwar wird der Prozess in Kapitel 5 genauer untersucht, doch sollen hier schon einige, für das Verständnis der Aufgabenstellung wichtige Punkte erläutert werden:

- Auf Wunsch des Kunden wird die Übersetzung der Texte nicht in einem zentralen Übersetzungsbüro, sondern von Vertretern des Kunden in den jeweiligen Ländern selbst vorgenommen. Grund dafür sind die vielen Fach- und vor allem unternehmensspezischen Begriffe, die in IUCCA verwendet werden. Dieser Prozess erzeugt jedoch einen hohen Kommunikationsaufwand zwischen den Entwicklern und den unterschiedlichen Übersetzern.
- Technisch wurde die Internationalisierbarkeit von IUCCA unter anderem durch die so genannten Java Properties-Dateien realisiert, die im Kapitel 4 erläutert werden. Diese Dateien sind nur bedingt für die Bearbeitung durch Personen geeignet, denen das technische Hintergrundwissen über den Aufbau und die Verwendung dieser Dateien fehlt. Daher werden die Properties-Dateien vor der Übersetzung in Excel-Tabellen importiert und nach der Übersetzung wieder in

Properties-Dateien umgewandelt. Dieser Prozess ist aufwendig und durchaus auch fehleranfällig.

- Die Übersetzer können in den Textressourcen nicht erkennen, in welchem Kontext die zu übersetzenden Texte später in der Anwendung in Erscheinung treten. Dies hat zur Folge, dass Texte nicht so übersetzt werden, wie es der Kontext erfordern würde.
- Stellt ein Kunde einen Übersetzungsfehler fest, nachdem die Übersetzung in das System eingepflegt wurde, so muss er Kontakt mit dem Entwicklerteam aufnehmen und einen Entwickler bitten den Fehler zu korrigieren oder sich die Textressourcen zuschicken lassen, um den Fehler selbst zu beheben. Wobei die im zweiten Punkt beschriebene Prozedur der Umwandlung der Properties-Dateien erneut durchlaufen werden muss. Dieses Problem wird sich mit der Einführung von IUCCA in Griechenland und Russland aufgrund der unterschiedlichen Schriftsysteme noch verschärfen, da dann die Entwickler Korrekturen kaum noch selbst durchführen können.
- Da in den einzelnen Ländern unterschiedliche IUCCA Versionen eingesetzt werden, erzeugt auch das Verwalten der in Übersetzung befindlichen Sprachressourcen erheblichen Aufwand.

Die oben beschriebenen Punkte erzeugen einen hohen Aufwand für die Übersetzung der Sprachressourcen von IUCCA. Daher wurde beschlossen, dass eine Anwendung geschaffen werden soll, die den bestehenden Prozess optimiert, indem möglichst viele Prozessschritte automatisiert und zentralisiert werden. Da es sich um eine internetbasierte Anwendung handeln soll, bekam das Projekt den Namen TROIA (Text Resources Online Integration and Administration).

3.2 Zielsetzung

Die Aufgabenstellung für diese Arbeit ist der Entwurf und die Realisierung einer Anwendung, die den im vorigen Kapitel beschriebenen Prozess für die Übersetzung von Sprachressourcen des IUCCA-Systems optimiert. Dabei sollen folgende Vorgaben beachtet bzw. realisiert werden:

- Entwurf eines optimierten Übersetzungsprozesses.
- Die Anwendung TROIA soll den neuen Übersetzungsprozess mit dem Vorgehensmodell für die IUCCA-Entwicklung synchronisieren.
- Der Kunde soll durch TROIA die Möglichkeit haben, die Übersetzung der Sprachressourcen parallel zum Test einer neuen Version durchzuführen.
- Die Versionisierung der Sprachressourcen soll durch TROIA unterstützt werden.

Das Ziel dieser Arbeit ist also die Entwicklung einer stabilen Anwendung, die sowohl von Anwendern als auch von den Entwicklern des IUCCA-Systems genutzt werden kann. Die Anwendung soll den Übersetzungsprozess der Sprachressourcen optimieren, dadurch das Entwicklungsteam entlasten und die Kundenzufriedenheit durch einen flexibleren Prozess erhöhen.

4 Internationalisierung von Anwendungen

Bevor auf die Konzeption und die Realisierung der in den letzten Kapiteln beschriebenen Anforderungen eingegangen wird, soll in diesem Kapitel ein allgemeiner Überblick über den Stand der Technik im Bezug auf die Internationalisierung von Software gegeben werden. Dabei werden die allgemeinen Definitionen, die Zeichensatzproblematik und die Möglichkeiten der Programmiersprache Java[2] betrachtet. Sowohl IUCCA als auch TROIA wurden mit dieser Programmiersprache realisiert.

4.1 Internationalisierung und Lokalisierung

Als erstes soll geklärt werden, was unter der Internationalisierung in der Softwareentwicklung verstanden wird. Dabei ist zwischen den Begriffen Locale, Internationalisierung und Lokalisierung zu unterscheiden.

Ein **Locale** (dt. etwa Schauplatz) beschreibt laut [Amshoff04] eine geographische, politische oder kulturelle Region, die eine eigene Sprache hat sowie spezifische Regeln für die Darstellung von Zahlen- oder Datumsangaben besitzt. Das Locale definiert also, an welchem Ort ein Programm ausgeführt werden soll.

Internationalisierung wird von [Shirah02] als ein Prozess definiert, durch den eine Anwendung so flexibel entworfen und implementiert wird, dass diese in verschiedenen Locales eingesetzt werden kann. Ein internationalisiertes Programm ist also in der Lage verschiedene Sprachen, aber auch Datums- bzw. Zeitformate, Währungen und weitere regional unterschiedliche Daten zu unterstützen und darzustellen, ohne das die Software geändert werden muss.

Internationalisierung wird in der Literatur oft als »i18n« abgekürzt. Diese Abkürzung bezieht sich auf das englische Wort »internationalization«, in dem zwischen dem ersten Buchstaben »i« und dem letzten Buchstaben »n« achtzehn Zeichen stehen. Diese Abkürzung ist auch in der deutschsprachigen Literatur weit verbreitet.

Lokalisierung[3] wird von [Shirah02] als ein Prozess definiert, bei dem eine Anwendung so entworfen und implementiert wird, dass diese einen bestimmten regional-, landes-, sprach-, kultur-, politik- oder geschäftsabhängigen Kontext unterstützt. Auch Lokalisierung (eng. localization) wird nach dem oben beschriebenen Verfahren als L10N abgekürzt.

---

[2] Java ist eine von Sun Microsystems entwickelte, objektorientierte Programmiersprache. Weitere Informationen findet man unter: http://java.sun.com .

[3] Engl. "localization", in diesem Kontext als "Anpassung an örtliche Besonderheiten" zu übersetzen.

Das bedeutet, dass eine internationalisierte Anwendung im Rahmen der Lokalisierung an ein konkretes Locale angepasst wird. Eine der Hauptaufgaben der Lokalisierung ist also die Übersetzung der für den Benutzer sichtbaren Texte in die jeweilige Sprache.

Ein internationalisiertes Programm hat laut [Green05] folgende Eigenschaften:

- Durch das Hinzufügen von lokalisierten Daten kann dieselbe Anwendung weltweit eingesetzt werden.
- Textelemente, wie z.B. die an der Benutzeroberfläche sichtbaren Text- oder Fehlermeldungen, sind nicht im eigentlichen Programmcode enthalten. Sie werden außerhalb des Quellcodes gespeichert und dynamisch eingebunden.
- Um neue Sprachen in das Programm einzubinden, muss das Programm nicht neu kompiliert werden.
- Kulturabhängige Daten, wie z.B. Datum, Uhrzeit oder Währung, werden in einem der Region und der Sprache des Endnutzers entsprechenden Format dargestellt.
- Das Programm kann schnell und einfach lokalisiert, also an die örtlichen Besonderheiten angepasst werden.

4.2 Internationalisierung und Zeichensätze

Wie bereits erwähnt ist eine der Hauptaufgaben der Internationalisierung das Darstellen von Texten in der Landessprache des jeweiligen Benutzers. Dafür müssen die Texte natürlich im Zuge der Lokalisierung für dieses Land zunächst übersetzt werden. Folglich stellt sich jedoch das Problem, dass die übersetzten Texte auch dargestellt werden müssen. Kritisch dabei ist, dass viele Sprachen Sonderzeichen bzw. eigene Schriftzeichen wie z.B. kyrillisch verwenden.

Da Computer grundsätzlich nur mit binären Zahlen arbeiten, müssen Buchstaben und sonstige Zeichen Bit- bzw. Byte-Werten zugeordnet werden, um sie speichern oder darstellen zu können. Diesen Vorgang nennt man Zeichenkodierung (engl. character encoding). Durch die Kodierung mehrerer Zeichen wird ein Zeichensatz (engl. character set) erzeugt, in dem Zeichen auf digitale Codes abgebildet werden.

Der in der Computertechnik bisher wichtigste Zeichensatz ist der ASCII-Zeichensatz[4]. Im ASCII-Zeichensatz wird jedes Zeichen mit 7 Bit kodiert. Dadurch sind also maximal 128 Zeichen möglich. Der ASCII umfasst, neben einigen Steuerzeichen, vor allem die in der englischen Sprache verwendeten Buchstaben des lateinischen Alphabets sowie die arabischen Ziffern. Schon viele in europäischen Sprachen verwendete Sonderzeichen, wie z.B. Umlaute, sind im ASCII-Zeichensatz nicht enthalten, von den Schriftzeichen anderer Alphabete ganz zu schweigen.

[4] ASCII - American Standard Code for Information Interchange.

Deswegen sind im Laufe der Zeit zahlreiche, gegenüber dem ASCII erweiterte Zeichensätze entwickelt worden, die mehr Zeichen umfassen. Einige dieser Zeichensätze sind propietär oder Industriestandards. Diese Zeichensätze werden auch Code Pages genannt. Weitere entsprechen nationalen oder ISO[5]-Normen, wie z.B. der westeuropäisch Zeichensatz ISO 8859-1 (ISO Latin-1), in dem auch alle deutschen Zeichen enthalten sind. ISO Latin-1 ist eine Erweiterung des ASCII und wird mit 8 Bit kodiert. Dies ermöglicht 256 verschiedene Zeichen. Allerdings ist das Euro-Zeichen nicht im ISO Latin 1 Zeichensatz enthalten, was immer wieder zu Problemen führt.

Durch die verschiedenen Standards kommt es oft zu Konflikten zwischen den Zeichensätzen. So kann es sein, dass ein asiatischer Zeichensatz den gleichen Wert auf ein anderes Zeichen abbildet als ein europäischer Zeichensatz. Dies kann in mehrsprachigen Anwendungen zu erheblichen Problemen mit der Darstellung von Zeichen führen. Aus diesem Grund kam es zu der Überlegung einen globalen Zeichensatz für alle Sprachen bzw. Schriften dieser Welt zu erstellen.

Ende der achtziger Jahre begannen zwei Organisationen unabhängig voneinander diesen globalen Zeichensatz zu spezifizieren. Die eine Organisation ist die International Organization for Standardization, eine Unterorganisation der UNO, mit ihrem ISO/ICE[6] 10646 Standard. Die andere ist das Unicode Konsortium mit dem Unicode Standard. Das Unicode Konsortium ist eine private Organisation verschiedener kommerzieller Unternehmen, akademischer Einrichtungen und privaten Anwendergruppen.

Anfang 1991 erkannten beide Organisationen, dass es wenig Sinn macht, zwei konkurrierende Standards ins Leben zu rufen. Man beschloss die bisherige Arbeit zusammenzuführen und den ISO/ICE 10646 Standard von nun an gemeinsam voranzutreiben.

[5] ISO - International Organization for Standardization

[6] ICE - International Electrotechnical Commision

4.3 Unicode

Laut [Unicode05] ist das Unicode Projekt zu hundert Prozent konform zum ISO/ICE 10646 Standard, bietet aber darüber hinaus noch wichtige Implementierungsalgorithmen und weitere nützliche semantische Informationen zu den kodierten Zeichen.

Der Unicode Standard wurde konzeptioniert um folgende Anforderungen zu realisieren (Unicode 2005):

- **Universal**: Der Umfang des Zeichensatzes sollte groß genug sein, um alle Zeichen zu umfassen, bei denen die Wahrscheinlichkeit besteht, dass sie im globalen Schriftverkehr verwendet werden. Außerdem sollen auch möglichst viele der gängigen Zeichensätze internationaler, nationaler und industrieller Standards unterstützt werden.
- **Effizient**: Texte können einfach analysiert werden. Verarbeitende Software muss sich keine Zustände merken oder auf bestimmte mehrdeutige Codes achten, wie es bei einigen anderen Zeichensätzen der Fall ist. Eindeutige Zeichencodes ermöglichen effizientes Sortieren, Suchen, Darstellen und Bearbeiten von Texten unterschiedlicher Sprache.
- **Eindeutig:** Jeder beliebige Unicode *code point* repräsentiert immer nur ein einziges Zeichen.

4.3.1 Schriftzeichen und Glyphen

Der Unicode Standard unterscheidet zwischen Schriftzeichen (engl. characters) und Glyphen bzw. Bildzeichen (engl. glyphs).

Unter Schriftzeichen versteht Unicode die abstrakten Repräsentanten der kleinsten Einheit einer geschriebenen Schrift. Sie stehen hauptsächlich, aber nicht ausschließlich, für Buchstaben, Interpunktionen und andere Symbole, welche die natürlichen Sprachen und technischen Notationen bilden.

Jedes Schriftzeichen wird durch so genannte *code points* identifiziert und hat einen eindeutigen, abstrakten Namen. Unicode *code points* können als »U+*n*« ausgedrückt werden, wobei *n* für vier bis sechs hexadezimale Ziffern steht. Diese hexadezimale Zahl ist der Wert, durch den ein Schriftzeichen in Unicode repräsentiert wird. So ist z.B. das Schriftzeichen »A« durch den *code point* U+0041 und den Namen »LATIN CAPITAL LETTER A« eindeutig identifizierbar.

Glyphen repräsentieren die Form, die Schriftzeichen annehmen können, wenn sie dargestellt oder angezeigt werden. Ein Menge von Glyphen bildet eine Schriftart, wie z.B. „Arial“. Die Gestaltung der Glyphen sowie das Entwickeln von Methoden zur Zuordnung von Glyphen zu Schriftzeichen ist die Aufgabe der Entwickler von Schriftsätzen und nicht Teil des Unicode Standards.

Glyphs	Unicode Characters	
A 𝐀 𝐴 𝑨 A 𝒜 𝓐 A	U+0041	LATIN CAPITAL LETTER A
a 𝐚 𝑎 𝒂 a 𝒶 𝓪 a	U+0061	LATIN SMALL LETTER A
fi fi	U+0066 + U+0069	LATIN SMALL LETTER F LATIN SMALL LETTER I
п *n* *ū*	U+043F	CYRILLIC SMALL LETTER PE
ه ھ ہ ۿ	U+0647	ARABIC LETTER HEH

Abbildung 2: Unterschied zwischen Schriftzeichen und Glyphen. [Unicode05]

Man kann also sagen, dass sich der Unicode Standard nur mit der logischen Repräsentation eines Schriftzeichens befasst. Die visuelle Darstellung eines Schriftzeichens durch Glyphen ist nicht Teil des Unicode Standards.

Die Abbildung 2 zeigt den Unterschied zwischen Glyphen und abstakten Schrift Zeichen. Man erkennt, dass das Zeichen »LATIN CAPITAL LETTER A« durch einen eindeutigen *code point* repräsentiert wird, aber durch viele unterschiedliche Glyphen dargestellt werden kann.

Wie bereits beschrieben stehen *code points* für den Zahlenwert, durch den ein Schriftzeichen in Unicode identifiziert wird. Der Bereich der möglichen Zahlenwerte, den Unicode bereitstellt, geht von 0 bis $10FFFF_{16}$. Dieser Bereich wird auch Coderaum (engl. code space) genannt. Das bedeutet, es stehen insgesamt 1 114 112 *code points* der Zuordnung von abstrakten Schriftzeichen zur Verfügung. Der Unicode Standard enthält, in der zum Zeitpunkt des Schreibens dieser Arbeit aktuellen Version 4.0, 96 382 Schriftzeichen. Diese decken nicht nur den Bedarf fast aller modernen Sprachen, sondern beinhalten auch die Schriftzeichen vieler historischer Sprachen, wie z.B. die Sumerische Keilschrift.

4.3.2 Unicode Planes

Der Unicode Coderaum ist in mehrere Ebenen (engl. planes) unterteilt. Diese Aufteilung hat sowohl historische als auch praktische Gründe.

Ursprünglich war der Coderaum von Unicode nur 16 Bit groß. Es zeigte sich aber bald, dass die dadurch maximal möglichen 65535 *code points* nicht ausreichend für die Erfassung aller modernen und historischen Schriftzeichen sind. Also wurde der Coderaum auf 32 Bit erweitert und das Konzept der Ebenen eingeführt, wobei jede Ebene ca. 65 000 *code points* beinhaltet. Die Ebenen dienen der Einteilung von Schriftzeichen in grobe Kategorien. Im Folgenden sollen die drei wichtigsten Ebenen kurz beschrieben werden.

Basic Multilingual Plane (BMP). Die BMP umfasst den Bereich von 0000_{16} bis $FFFF_{16}$. Dies entspricht dem ursprünglichen Umfang von Unicode von 65535 *code points*. In der BMP werden die meisten der heute gebräuchlichen Schriftzeichen dieser Welt kodiert. Auf dieser Ebene sind auch ein Großteil der CJK[7]-Schriftzeichen erfasst. Weiterhin entsprechen die ersten 256 der 65.536 *code points* den Zeichen des Zeichensatzes ISO 8859-1, der eine Erweiterung von ASCII ist.

Supplementary Multilingual Plane (SMP). Die SMP umfasst den Bereich von 10000_{16} bis $1FFFF_{16.}$ Diese Ebene ist für die Erfassung von selten genutzten historischen Schriften oder speziellen Notationen, die nicht in die BMP aufgenommen wurden, gedacht. In Unicode 4.0 sind in dieser Ebene erst wenige Zeichen erfasst, allerdings existieren noch viele historischen Schriftsysteme, die bisher nicht in Unicode erfasst wurden. Diese Schriften sollen in Zukunft in die SMP aufgenommen werden.

Supplementary Ideographic Plane (SIP). Die SIP umfasst den Bereich von 20000_{16} bis $2FFFF_{16}$. Diese Ebene wurde all den CJK-Schriftzeichen zugewiesen, die nicht mehr in die BMP gepasst haben. Die meisten dieser Schriftzeichen werden nur sehr selten benutzt oder sind nur noch von historischem Interesse. Einige wenige werden allerdings auch heute noch häufig verwendet.

Die Zeichen, die in den zusätzlichen Ebenen außerhalb der BMP erfasst werden, nennt man auch *supplementary characters* (dt. Zusatzzeichen). In Unicode 4.0 sind bislang 96.382 Codes individuellen Zeichen zugeordnet worden. Das entspricht in etwa erst 9% des gesamten Coderaumes.

4.3.3 Unicode Encoding Forms

Ein Computer behandelt Zahlen nicht einfach als abstrakte mathematische Objekte, sondern als eine Kombination aus Bitketten mit fester Länge, wie z.B. Bytes oder 32-Bit-Wörtern. Diese Bitketten werden auch *code units* (dt. Codeblöcke) genannt. Dies ist zu beachten, wenn festgelegt werden soll, wie *code points* auf *code units* abgebildet werden sollen.

Da Computersysteme gewöhnlich Blöcke von 8 Bit (= 1 Byte), 16 Bit oder 32 Bit verwenden, bietet der Unicode Standard drei voneinander unabhängige Kodierungsarten (engl. encoding forms). Diese Kodierungsarten benutzen entweder 8-Bit, 16 Bit oder 32 Bit lange *code units* um Unicode Schriftzeichen zu kodieren. Entsprechend werden diese Kodierungsarten UTF[8]-8, UTF-16 oder UTF-32 genannt. Laut [Unicode05] kann jede dieser Kodierungsarten den gesamten Bereich der Unicode-Schriftzeichen abbilden.

Dies ist eine wichtige Aussage, da man bei der Recherche im Internet immer wieder auf die Ansicht stößt, dass mit UTF-8 weniger Zeichen dargestellt werden können, als mit UTF-16 oder UTF-32.

---

[7] CJK – Chinesisch Japanisch Koreanisch

Um zu verstehen, wie es möglich ist, mit dem acht Bit basierten UTF-8 die gleiche Anzahl von Schriftzeichen darzustellen, wie mit UTF-32, müssen die einzelnen Transformationsformate genauer betrachtet werden.

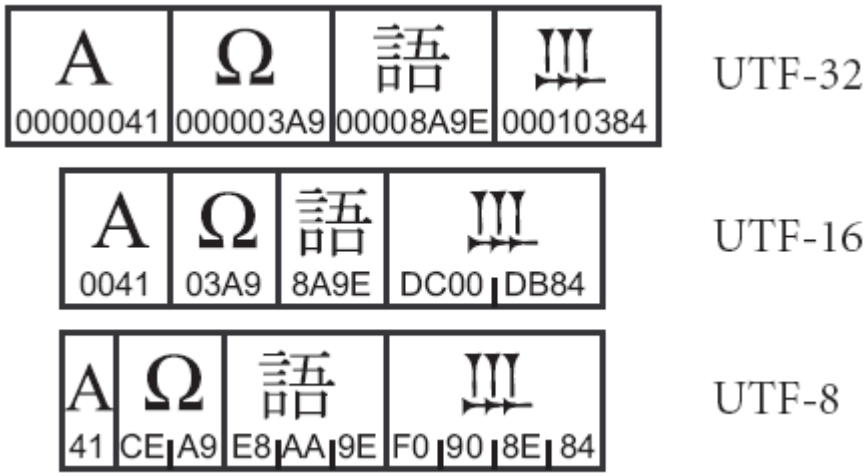

Abbildung 3: Unicode Kodierungsarten. [Unicode05]

4.3.4 UTF-32

UTF-32 ist die einfachste Kodierungsart. Jeder Unicode *code point* wird direkt durch eine einzige 32-Bit-*code unit* abgebildet. Daher besteht bei UTF-32 eine 1:1 Beziehung zwischen *code units* und Unicode *code points*. So hat das in Abbildung 3 dargestellte sumerische Keilschriftzeichen den Unicodewert U+10384 und wird in UFT-32 auch durch den Wert »00010384« dargestellt. Wie man in der Abbildung auch sehen kann, wird dieser Unicode *code point* in UTF-16 und UTF-8 durch *code units* mit anderen Werten repräsentiert.

Theoretisch ist es möglich, mit 32-Bit 4 294 967 296 *code points* darzustellen. Allerdings wurde UTF-32 auf die Repräsentation von *code points* aus dem Bereich 0 bis $10FFFF_{16}$ beschränkt. Dies entspricht den bereits erwähnten 1 114 112 *code points* des Unicode *code space*. Die Beschränkung auf 1 114 112 *code points* garantiert die Kompatibilität zu UTF-16 und UTF-8.

4.3.5 UTF-16

In der UTF-16 Kodierungsart werden alle *code points* im Bereich zwischen 0 und $FFFF_{16}$ als eine einzelne 16-Bit *code unit* dargestellt. Dieser Bereich entspricht der BMP und beinhaltet die gängigen aktuellen Schriftsysteme dieser Welt. Der Bereich von 10000_{16} bis $10FFFF_{16}$ wird durch zwei 16-Bit *code units* abgebildet. Diese Codeblockpaare werden als *surrogate pairs* (dt. Ersatzpaare) bezeichnet. Für die UTF-16

[8] UTF – Unicode Transformation Format.

Kodierung der *supplementary characters,* also der Zeichen die nicht in der BMP erfasst wurden, ist im Unicode Coderaum ein separater Bereich reserviert. Dieser Bereich ist ausschließlich für die Werte der UTF-16 *surrogate pairs* vorgesehen. Er wurde in die »*high-surrogates range«*“ und die »low-surrogates range« unterteilt. In einem *surrogate pair* stammt der Wert der ersten *code unit* aus der »high-surrogates range« und der Wert der zweiten *code unit* aus der »low-surrogates range«. Dadurch kann jederzeit bestimmt werden, ob es sich bei einer *code unit* um eine einzelne *code unit* oder die erste bzw. zweite *code unit* eines Ersatzpaares handelt.

In der Abbildung 3 erkennt man, dass die ersten drei Zeichen als jeweils nur eine *code unit* dargestellt werden. So hat die *code unit* für das chinesische Schriftzeichen den Wert »8A9E«. Dies entspricht dem Unicode *code point* U+8A9E. Das sumerische Schriftzeichen wird allerdings durch zwei *code units* dargestellt und die UTF-16 Werte entsprechen nicht direkt dem Unicode *code point*. Um auf den Unicode *code point* zu kommen, muss der UFT-16 Wert transformiert werden.

4.3.6 UTF-8

Die UTF-8 Kodierung wurde entwickelt, um den Ansprüchen von byte-orientierten und ASCII-basierten Systemen zu genügen. Wie bereits beschrieben, integriert Unicode viele bereits existierende Zeichensätze, darunter auch ASCII. So sind die Unicode *code points* aus dem Bereich von 0 bis $7F_{16}$ (0 bis 127) identisch mit den ASCII-Codes. In der UTF-8-Kodierung werden Codes aus diesem Bereich als jeweils ein Byte wiedergegeben. Somit sind alle Daten, die ausschließlich echte ASCII-Zeichen verwenden, sowohl in der ASCII- als auch in der Unicodedarstellung identisch.

Unicode *code points,* die einen Wert größer als 127 haben, werden in der UTF-8-Kodierung zu Byte-Ketten mit der Länge zwei bis vier. In der Tabelle 1 kann man sehen, welche Unicodebereiche mit wie vielen Bytes kodiert werden. Das erste Byte eines UTF-8 kodierten Zeichens nennt man dabei Startbyte. Weitere Bytes nennt man Folgebytes. Startbytes beginnen also mit der Bitfolge 11 oder einem 0-Bit, während Folgebytes immer mit der Bitfolge 10 beginnen.

Tabelle 1: Unicode-Bereich und UTF-8-Kodierung Quelle: [WikiUTF05]

Unicode-Bereich	**UTF-8-Kodierung**
0000 0000 - 0000 007F	0xxxxxxx
0000 0080 - 0000 07FF	110xxxxx 10xxxxxx
0000 0800 - 0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000 - 0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Betrachtet man die Bitfolgen etwas genauer, erkennt man die Arbeitsweise von UTF-8 [WikiUTF05]:

- Ist das höchste Bit des ersten Byte 0, handelt es sich um ein gewöhnliches ASCII-Zeichen. Dadurch sind alle ASCII-Dokumente automatisch aufwärtskompatibel zu UTF-8.
- Ist das höchste Bit des ersten Byte 1, handelt es sich um ein Mehrbytezeichen, also ein Unicode-Zeichen mit einem *code point,* dessen Wert größer als 127 ist.
- Sind die höchsten beiden Bits des ersten Byte 11, handelt es sich um das Start-Byte eines Mehrbytezeichens. Sind sie 10, um ein Folge-Byte.
- Bei den Startbytes gibt die Anzahl der 1-Bits vor dem ersten 0-Bit die gesamte Bytezahl des UFT-8 kodierten Unicodezeichens an.
- Start-Bytes (0xxx xxxx oder 11xx xxxx) und Folge-Bytes (10xx xxxx) lassen sich eindeutig voneinander unterscheiden. Somit kann ein Byte-Strom auch in der Mitte gelesen werden, ohne dass es Probleme mit der Dekodierung gibt. Bytes die mit 10 beginnen werden einfach übersprungen, bis ein Byte gefunden wird, das mit 0 oder 11 beginnt. Könnten Startbytes und Folgebytes nicht eindeutig voneinander unterschieden werden, wäre das Lesen eines UTF-8 Datenstroms, dessen Beginn unbekannt ist, unter Umständen nicht möglich.

4.3.7 Vergleich zwischen UTF-8, UTF-16 und UTF-32

Das UTF-32-Kodierungssystem verwendet für jedes Zeichen einen Block mit der festen Länge von 32 Bit. Daher ist der Speicherbedarf von UTF-32 in den meisten Fällen doppelt so hoch, wie bei UTF-16 und im Fall von ASCII-Texten viermal so hoch, wie der von UTF-8. Der Vorteil ist allerdings, dass das Verarbeiten von Texten sehr effizient ist, da nie geprüft werden muss, ob ein Zeichen mit einem oder mehreren *code units* kodiert ist.

UTF-16 verwendet für alle Zeichen, die in der BMP enthalten sind, eine *code unit* mit der Länge von 16 Bit. Nur die Zeichen aus den höheren Ebenen werden mit zwei *code units* kodiert. Das bedeutet, dass fast alle Texte die in modernen Schriftsystemen abgefasst sind, mit einem 16 Bit-Block kodiert sind. Die meisten Anwendungen können also davon ausgehen, dass feste Blöcke verwendet werden. Daher ist das Verarbeiten von Texten bei UTF-16 ähnlich effektiv, wie bei UTF-32, wobei UTF-16 in den meisten Fällen nur den halben Speicherbedarf von UTF-32 benötigt. UTF-16 stellt also einen guten Kompromiss zwischen Speicherbedarf und Performance dar.

UTF-8 verwendet für die Kodierung eine variable Anzahl von 8 Bit-Blöcken (Bytes). Daher ist die Performance der Textverarbeitung nicht so gut, wie bei den anderen Kodierungssystemen. Es muss immer geprüft werden, ob es sich bei dem aktuellen Byte um ein Start- oder Endbyte handelt. Allerdings benötigt UTF-8, zumindest für ASCII-Daten, den geringsten Speicherbedarf. Die meisten der anderen, in der BMP enthaltenen Zeichen, benötigen, wie auch bei UTF-16, 16 Bit Speicher. Eine Ausnahme bilden

jedoch die fernöstlichen Schriften, wie Chinesisch, Japanisch und Koreanisch. Diese Schriften sind auch in der BMP enthalten und werden von UTF-16 mit einem 16 Bit Block dargestellt. Wie in Abbildung 3 ersichtlich, kodiert UTF-8 diese Zeichen hingegen mit drei Bytes, also 24 Bit, und benötigt in diesem Fall mehr Speicher als UTF-16.

Der große Vorteil von UTF-8 ist jedoch die Kompatibilität zu byteorientierten und ASCII-basierten Systemen. Daher ist UFT-8 das momentan populärste und vor allem im Internet am weitesten verbreitete Unicode-Kodierungssystem.

4.3.8 Sicherheit in Unicode

Wie in vielen anderen Standards und Protokollen in der Computertechnik hat auch die Unicode Spezifikation Schwachstellen, die ausgenutzt werden können um die Sicherheitsvorkehrungen von Systemen zu unterlaufen oder Anwender zu täuschen.

4.3.8.1 Der UTF-8 Exploit[9]

InUTF-8 ist es möglich einen Unicode *code point* in mehreren Formen darzustellen. So wird der *code point* U+0000 normalerweise in einem Byte mit dem Binärwert »0 0000000« dargestellt. Darüber hinaus kann dieser *code point* aber auch durch zwei, drei oder vier Bytes ausgedrückt werden. Die zwei Byte-Darstellung würde folgendermaßen aussehen: »110 00000 10 000000« (siehe Tabelle 1). Diese Darstellungsformen nennt man *non-shortest forms* (dt. nicht kürzeste Form).

Bis einschließlich Unicode-Version 3.0 war laut [Davis05] zwar das generieren dieser *non-shortest forms* nicht erlaubt, die Interpretation dieser Bytefolgen war hingegen gestattet, denn dadurch wurde der Algorithmus zum Umwandeln von UTF-8 in UTF-16 vereinfacht.

Sollte nun Firewalls oder andere Sicherheitssysteme Texte auf bestimmte Zeichen analysieren und dabei nur die erlaubten UTF-8 Repräsentationen dieser Zeichen berücksichtigen, können *non-shortest forms* dieser Zeichen ungehindert passieren. Wenn weiterhin das zu schützende System diese *non-shortest forms* interpretiert, wären die Sicherheitsmaßnahmen unterlaufen und ein direkte Interaktion mit dem System möglich.

In der Unicode-Version 3.1 wurde die Interpretation von *non-shortest forms* untersagt. Allerdings dürfte es noch viele Systeme geben, die durch diesen *exploit* angreifbar sind.

4.3.8.2 Visual Spoofing

Unter *Visual Spoofing* (dt. optische Manipulation) versteht man nach [Davis05] das Täuschen eines Anwenders mit Hilfe optisch ähnlicher Darstellungen, durch die er zu sicherheitskritischen Handlungen bewegt werden soll.

---

[9] Unter *exploit* (dt. ausnutzen) versteht man den Angriff auf ein Computersystem unter Ausnutzung einer Sicherheitslücke.

Visual Spoofing ist jedoch keine Effekt der nur Unicode betrifft, auch mit ASCII-Zeichen kann *Visual Spoofing* betrieben werden. So lautet der Domainname der Firma Intel »intel.com«, allerdings kann diese Domäne auch als »inteI.com« geschrieben werden. Optisch sehen die beiden Zeichenketten gleich aus, allerdings wurde bei der zweiten Variante der Kleinbuchstabe für »L« durch den Großbuchstaben »I« ersetzt.

Das *Visual Spoofing* basiert also auf optisch ähnlich wirkenden Zeichen. Obwohl *Visual Spoofing* auch ohne Unicode betrieben werden kann, erhöht der riesige Zeichenvorrat von Unicode die Täuschungsmöglichkeiten um ein vielfaches.

Durch die Einführung der *Internationalized Domain Names*[10] (IDN) ist es nun erlaubt Unicodezeichen in Domainnamen zu verwenden. Dadurch wird aber auch das *Cross Script Spoofing* ermöglicht, bei dem in einem Begriff Zeichen unterschiedlicher Zeichensätze verwendet werden.

Tabelle 2: Cross Script Spoofing

	Begriff	**UTF-16**	**IDNA**
1	top.com	0074 **03BF** 0070 002E 0063 006F 006D	xn - - tp - jbc.com
2	top.com	0074 **006F** 0070 002E 0063 006F 006D	top.com

In Tabelle 2 wird ein Beispiel *Cross Script Spoofing* gezeigt. Für einen Anwender sieht der dargestellte Domainname »top.com« in beiden Fällen völlig identisch aus. Er hat keine Möglichkeit zu erkennen, das im Beispiel 1 der griechische Buchstabe »Omicron« und in Beispiel 2 der lateinische Buchstabe »o« verwendet wurde.

Die hier angeführten Beispiele zeigen, dass die Verbreitung von Unicode nicht nur Vorteile hat, sondern auch zu erheblichen Sicherheitsrisiken führen kann. Es ist zu erwarten, das in Zukunft noch mehr durch Unicode verursachte Schwachstellen und Täuschungsmöglichkeiten gefunden werden. Einen guten Überblick über die aktuellen Sicherheitsaspekte von Unicode findet man in dem Dokument von [Davis05], aus dem auch die hier gezeigten Beispiele stammen.

4.3.9 Zusammenfassung Unicode

Unicode ist ein international anerkannter Standard, dessen Ziel die Kodierung aller weltweit jemals verwendeten Schriftzeichen in einem einzigen Zeichensatz ist. Dabei ist Unicode zu vielen der bereits existierenden Zeichensätze kompatibel.

Unicode bietet mit UTF-8, UTF-16 und UTF-32 Kodierungssysteme, die alle drei den gesamten Zeichenvorrat von Unicode repräsentieren können, aber unterschiedliche Eigenschaften in Bezug auf Speicherverbrauch und Effizienz der Verarbeitung haben.

[10] Siehe auch http://de.wikipedia.org/wiki/Internationalized_Domain_Name

Durch die Einführung von Unicode in ein System können Sicherheitsrisiken entstehen, deshalb muss geprüft werden, ob und wie sich Unicode auf die Sicherheit dieses Systems auswirkt.

Ein weiteres Problem mit Unicode ist die Darstellung der Zeichen an einem Computer. Denn Unicode definiert nur Zeichenwerte und Eigenschaften von Zeichen, aber es enthält ebenso wenig Angaben über die Darstellung eines Zeichens, wie herkömmliche Zeichensätze. Dazu sind am Computer Schnittstellen in Form von Schriftarten erforderlich. Die klassischen Computerschriftarten sind dazu jedoch nur bedingt geeignet, da sie sich weitgehend an bestimmten Zeichensätzen orientieren. Neue, Unicode-orientierte Schriftarten verbreiten sich allmählich.

4.4 Internationalisierung mit Java

Nachdem in den letzten Kapiteln allgemeine Aspekte der Internationalisierung beleuchtet wurden, wird in diesem Kapitel auf die Hilfsmittel eingegangen, welche die Programmiersprache Java dem Entwickler für die Internationalisierung zur Verfügung stellt.

4.4.1 Java und Unicode

Die Programmiersprache Java benutzt Unicode als internen Zeichensatz. In der Version 1.4 der „Java 2 Standard Edition“ (J2SE) wurde der Unicode Standard 3.0 unterstützt. In der aktuellen Version 1.5 unterstützt die J2SE den Unicode Standard 4.0.

Die Unicodeunterstützung in Java wird durch den zwei Bytes (16 Bit) langen, primitiven Datentyp `char` realisiert. In Java 1.4 enthält ein `char` einen UTF-16 *code point*. Deshalb können ausschließlich Zeichen aus der *Basic Multilingual Plane* repräsentiert werden. Zeichen aus den *Supplementary Planes*, die alle 32 Bit benötigen, können nicht dargestellt werden. Das hat den Nachteil, dass in einigen fernöstlichen Sprachen nicht alle benötigten Schriftzeichen zur Verfügung stehen. Mit der Anpassung an den Unicode Standard 4.0 in J2SE 1.5 wurde dieser Mangel behoben. Zeichen aus den *Supplementary Planes* werden nun durch einen Array von zwei `chars` repräsentiert, wobei ein `char` nun UTF-16 *code units* enthält, statt wie bisher *code points* [LiOk05]. Da in UFT-16 die Werte für *code units* und *code points* bei der Darstellung von Zeichen aus der BMP identisch sind, hat diese Erweiterung auf die meisten bestehenden Programme keinen Einfluss. Allerdings wurden in allen auf dem Datentyp `char` basierenden Klassen, wie z.B. `Character`, `String` und `StringBuffer`, zusätzliche *low level* APIs eingefügt, die diesen Unterschied beachten und die Verarbeitung von *supplementary characters* ermöglichen.

Um auf Plattformen, die Unicode nicht unterstützen, ebenfalls Unicodezeichen eingeben zu können, bietet Java Unicode-Escapezeichen an. Diese Escapesequenzen werden in Form von »\u« gefolgt von vier Hexadezimalzahlen dargestellt. Die Hexadezi-

malzahlen repräsentieren den Wert einer UFT-16 *code unit.* So sind die beiden folgenden Ausdrücke gleichwertig:

```
char c1 = ‚a’ ;
char c2 = ‚\u0061’;
```

Seit J2SE 1.5 können nun auch *supplementary charaters* durch zwei UTF-16 *code units* ausgedrückt werden [LiOk05]. So könnte das sumerische Schiftzeichen aus Abbildung 3 in Unicode-Escapezeichen als »\udc00 \udb84« dargestellt werden. Außerdem wurden auch Escape-Zeichen für *code points* eingeführt. Diese haben die Form »\Uxxxxxx«, wobei das große »U« angibt, dass es sich um einen *code point* handelt. Dem Großbuchstaben folgen dann sechs Hexadezimalzahlen, die den Unicodewert repräsentieren.

4.4.2 Java Klassen zur Internationalisierung

In diesem Abschnitt wird auf die wichtigsten Klassen eingegangen, die durch die Java API für die Internationalisierung bereitgestellt werden.

4.4.2.1 Die Klasse Locale

Ausgangspunkt der Lokalisierung eines Java-Programms ist die Klasse `java.util.Locale`. Jede Instanz dieser Klasse identifiziert eine bestimmte geografische, kulturelle oder politische Region auf der Erde inklusive der Sprache, die dort üblicherweise verwendet wird. Ein Objekt kann wie folgt erzeugt werden:

```
public Locale(String language, String country);
```

Das erste Argument ist ein String mit dem Code für die Sprache, wie er im ISO-Standard 639[11] definiert ist. Der Sprachcode wird klein geschrieben und lautet z.B. »en« für englisch, »fr« für französisch oder »de« für deutsch. Das zweite Argument gibt das Land gemäß dem ISO-Standard 3166[12] an. Hier werden stets große Buchstaben verwendet. So stehen beispielsweise »US« für die USA, »GB« für England, »FR« für Frankreich und "DE" für Deutschland.

Der Länderteil ist optional. Wird an seiner Stelle ein Leerstring übergeben, repräsentiert die `Locale` lediglich eine Sprache ohne spezifisches Land. So steht »en« für die englische Sprache, wie sie nicht nur in England, sondern auch in den USA oder Kanada verständlich wäre. Sollen hingegen auf kanadische Besonderheiten eingegangen werden, ist als Land »CA« zu ergänzen. Für den französischsprachigen Teil von Kanada würde dagegen ein `Locale` aus »fr« und »CA« gebildet werden.

Neben Sprach- und Länderkennung kann ein `Locale` einen dritten Parameter haben. Dieser wird als Variante bezeichnet und ist ebenfalls optional. Die Verwendung sowie die möglichen Werte dieses Parameters sind nicht standardisiert. Der Parameter kann

[11] ISO Language Code: www.ics.uci.edu/pup/ietf/http/related/iso639.txt
[12] ISO Country Code: www.iso.org/en/prods-services/iso3166ma/

z.B. verwendet werden um Betriebssysteme oder Konfigurationen von einander zu unterscheiden. Laut [Amshoff04] bietet die Variante eine elegante Möglichkeit, im Rahmen der Mandantenfähigkeit einen bestimmten Mandanten zu kodieren.

Die Klasse `Locale` bietet mit der statischen Methode `getDefault()` einem Programm die Möglichkeit, zur Laufzeit zu ermitteln, in welchem Land es läuft und welche Sprache dort gesprochen wird.

4.4.2.2 Datum und Zeitzonen

Die korrekte Darstellung von Datumsangaben unter Berücksichtigung der Zeitzonen sind wichtige Aspekte bei der Internationalisierung von Anwendungen. Auch hier bietet Java Klassen, die das Erzeugen von lokalisierten Datumsangaben unterstützen.

Ursprünglich war die Klasse `java.util.Date` dazu gedacht, alle datums- und zeitbezogenen Operationen zu handhaben. Doch wurde sie laut [Shirah02] auf Grund interner Mängel darauf beschränkt, einen bestimmten Zeitpunkt zu repräsentieren. Im JDK 1.1 wurden die abstrakte Klasse `java.util.Calendar` und deren konkrete Subklasse `java.util.GregorianCalendar` eingeführt, um die Defizite der Klasse `java.util.Date` auszugleichen.

Des Weiteren gibt es noch die abstrakte Klasse `java.util.TimeZone` und ihre konkrete Subklasse `java.util.SimpleTimeZone`. Diese Klassen verwalten die, von der *Coordinated Universal Time* (UTC) abgeleiteten Zeitzonen. Die UTC ist die Nachfolgerin der mittleren Greenwichzeit (GMT). Ein Objekt der Klasse `TimeZone` wird wie folgt erzeugt:

```
TimeZone tz = TimeZone.getDefault();
```

Mit der Methode `getDefault()` kann das Programm ermitteln, in welcher Zeitzone es eingesetzt wird.

4.4.2.3 Locale-abhängiges Formatieren und Verarbeiten von Werten

Die Darstellung von Zahlen-, Währungs-, Datums- und Zeitwerten wird stark durch regionale und kulturelle Besonderheiten geprägt. Dies muss sowohl bei der Lokalisierung der Daten als auch bei der Internationalisierung einer Anwendung berücksichtigt werden. Java bietet für das Formatieren und Parsen von `Locale`-abhängigen Daten das Paket `java.text`. Die abstrakte Klasse `java.text.Format` als auch alle ihre Subklassen bieten Methoden, um Werte `Locale`-abhängig zu manipulieren. Darüber hinaus bietet die abstrakte Klasse `java.text.Collator` und ihre Subklassen `Locale`-abhängige Stringoperationen.

4.4.2.4 Laden von Ressourcen

Die Definition eines internationalisierten Programms von [Green05] sagt aus, dass ein internationalisiertes Programm lokalisiert werden kann, ohne es neu kompilieren zu müssen. Das bedeutet, dass alle evtl. zu lokalisierenden Daten nicht im eigentlichen Programmcode stehen dürfen, sondern in spezielle Dateien ausgelagert werden müssen. Diese zusätzlichen Dateien werden als *Ressourcen* bezeichnet und müssen zusammen mit dem Programm ausgeliefert werden. Die meisten von ihnen müssen lokalisiert werden.

Java stellt seit der Version 1.1 mit der Klasse `java.util.ResourceBundle` ein universelles Hilfsmittel zur Verwendung derartiger Ressourcen zur Verfügung. Ein `ResourceBundle` ist also ein Java-Objekt, das eine Sammlung von Ressourcen zusammenfasst und auf Anfrage einzeln zur Verfügung stellt. Jede Ressource hat einen eindeutigen und für alle unterstützten Sprachvarianten identischen Namen, mit dem darauf zugegriffen werden kann. Ein konkretes `ResourceBundle` ist stets sprachabhängig und enthält nur die Ressourcen für eine bestimmte *Locale*.

[Shirah02] beschreibt die Vorteile des `ResourceBundle` wie folgt:

- Für das Finden und Laden von Ressourcen wird der Java *Classloader* verwendet. Dies ist der selbe Mechanismus, wie er auch beim Laden von Javaklassen benutzt wird. Durch die Verwendung des *Classloaders* wird kein spezieller I/O[14] Code benötigt.
- Das `ResourceBundle` »weiß«, wie es die Hierarchie nach `Locale`-abhängigen Instanzen durchsuchen muss.
- Wenn die Ressource einer speziellen Instanz nicht gefunden wurde, wird die Ressource einer allgemeineren Instanz verwendet.

Die Klasse `ResourceBundle` verwendet Schlüssel-/Wert-Paare, wobei die Werte die zu lokalisierenden Daten darstellen, die über die Schlüssel gefunden werden können. Damit ist ein `ResourceBundle` zunächst nicht viel mehr als eine Schlüssel-/Wert-`Collection`, wie beispielsweise die Klassen `java.util.Hashtable` oder `java.util.HashMap`. Das Besondere an einem `ResourceBundle` liegt in der Tatsache, dass der Name des `ResourceBundle` das `Locale` angibt, dessen Daten es enthält. Während der vordere Teil den Basisnamen darstellt und für alle lokalisierten Varianten gleich ist, ist der hintere Teil `Locale`-spezifisch und gibt Sprache, Land und evtl. Variante an. Lautet der Basisname beispielsweise »messages«, wären »messages_fr« und »messages_en_US« die Namen für die französische und US-englische Variante.

[14] I/O steht für Input (dt. Eingabe) und Output (dt. Ausgabe). I/O steht für den Prozess, der Ein- und Ausgabe von Daten in Computerprogrammen.

Das `ResourceBundle` stellt eine statische Methode `getBundle` zur Verfügung, mit der zu einem Basisnamen und einer vorgegebenen `Locale` ein `ResourceBundle` beschafft werden kann. Diese gibt es in unterschiedlichen Ausprägungen:

```
public static final ResourceBundle getBundle(String baseName)
                                        throws MissingResourceException

public static final ResourceBundle getBundle(String baseName, Locale locale)
                                        throws MissingResourceException
```

Die erste Variante besorgt ein `ResourceBundle` für das aktuelle Default-`Locale`. Die zweite Variante liefert ein `ResourceBundle` für das als Argument angegebene `Locale`. In beiden Fällen wird wie folgt vorgegangen:

- Zunächst wird versucht, eine genau zur gewünschten `Locale` passende Ressourceklasse zu finden.
- Ist eine solche nicht vorhanden, wird der Ressourcenname schrittweise verallgemeinert. Dabei wird zunächst die Variante, dann der Länder- und schließlich der Sprachteil entfernt, bis nur noch der Basisname übrig bleibt.
- Falls immer noch keine passende Ressourcenklasse gefunden wurde, wird derselbe Ablauf für das Default- `Locale` wiederholt.

Ist beispielsweise Deutschland das Default-`Locale` und soll die Ressource »MyTextResource« für Frankreich beschafft werden, sucht `getBundle` nacheinander nach folgenden Klassen:

MyTextResource_fr_FR

MyTextResource_fr

MyTextResource_de_DE

MyTextResource_de

MyTextResource

Die Suche bricht ab, wenn die erste passende Klasse gefunden wurde. Kann überhaupt keine passende Ressource gefunden werden, so löst die Methode `getBundle` eine `MissingResourceExeption` aus. Dies ist auch der Fall, wenn die gefundene Klasse nicht aus `ResourceBundle` abgeleitet wurde.

Damit die Suche nach Ressourcen richtig funktioniert, müssen zu einer Ressource auch stets die allgemeineren Ressourcen vorhanden sein.

Man kann eigene Ressourcenklassen erstellen, die von `ResourceBundle` abgeleitet werden müssen und beiden in der Basisklasse abstrakten Methoden, `getKeys` und `handleGetObject` überschreiben.

4.4.2.5 Die Klasse PropertyResourceBundle

Für die Lokalisierung von einfachen Texten stellt die Java API die Klasse `PropertyResourceBundle` zur Verfügung, die von `ResourceBundle` abgeleitet ist. Der Unterschied liegt in der Speicherung der Daten. Die `PropertyResourceBundle` werden von ».properties«-Dateien unterstützt, die den Vorgaben der Klasse `java.util.Properties` genügen. Laut [Shirah02] sind dies folgende Punkte:

- Die Datei verwendet zur Zeichenkodierung den Standard ISO 8859-1.
- Die Datei enthält Schlüssel-Wert-Paare in Form von `schlüssel=wert`.
- Zeilen, die mit dem Zeichen »#« beginnen sind Kommentare.
- Zusätzlich zu der oben beschriebenen Namenskonvention für Ressourcen wird die Dateiendung ».properties« verwendet. So hat eine Properties-Datei z.B. den Namen »messages_de_DE.properties«.

Die Klasse `PropertyResourceBundle` liest also die Properties-Datei und speichert alle darin gefundenen Schlüssel-Wert-Paare ab. Anschließend können diese als gewöhnliche Textressourcen vom Programm verwendet werden.

Ein wichtiger Aspekt von Properties-Dateien ist deren Beschränkung auf den Zeichensatz ISO-Latin1. Das bedeutet, dass alle Zeichen, die in diesem Zeichensatz nicht enthalten sind, als Java Unicode-Escape-Zeichen eingegeben werden müssen. So enthält z.B. die Properties Datei »numbers_de_DE.properties« das Schlüssel-Wert-Paar:

```
one=Eins
```

In der russischen Properties-Datei würde dieser Eintrag folgendermaßen aussehen:

```
one=\u041E\u0438\u043D
```

Diese Einschränkung soll gewährleisten, dass die Datei mit jedem herkömmlichen Editor bearbeitet werden kann. Zwar muss man die Konvertierung in Java Unicode-Escape-Zeichen nicht von Hand durchführen, da das JDK[15] Tools anbietet, mit denen zwischen verschiedenen Zeichensätzen konvertiert werden kann. Dennoch hat diese Einschränkung zur Folge, dass das Bearbeiten von Properties-Dateien für `Locales`, die andere Zeichensätze als ISO-Latin 1 verwenden, sehr mühsam und benutzerunfreundlich ist. Wie in den folgenden Kapiteln gezeigt wird, ist dies eine erhebliche Einschränkung.

Wenn man beschließt, Java Properties-Dateien für die Internationalisierung einer Anwendung zu verwenden, sollte man sich schon zu Beginn eines Projektes überlegen, wie und vor allem, von wem die Inhalte der Properties-Dateien lokalisiert werden sollen und entsprechende Prozesse aufsetzen. Auch können bei größeren Anwendungen, an denen mehrere Entwickler arbeiten, die Pflege und die Verwaltung der Schlüssel in den Properties zu einer nicht zu unterschätzenden Herausforderung heranwachsen.

---

[15] JDK – Java Development Kit

Seit Java 1.5 können die Properties auch im XML-Format gespeichert und geladen werden. Da XML-Dateien standardmäßig UTF-8 als Zeichenkodierung nutzen, können in XML-basierten Properties-Dateien beliebige Unicode-Zeichen verwendet werden. Dies erhöht die Benutzerfreundlichkeit ungemein.

5 Die IUCCA-Anwendung

In Kapitel 2.2 wurde bereits ein kleiner Überblick über das Einsatzgebiet und die Funktionsweise von IUCCA gegeben. Da die Funktionalität von IUCCA keinen direkten Einfluss auf die Aufgabenstellung dieser Diplomarbeit hat, wird hier nicht tiefer darauf eingegangen. In diesem Kapitel werden der Aufbau des IUCCA-Systems sowie einige der verwendeten Internationalisierungstechniken betrachtet. Anschließend wird der Entwicklungsprozess für IUCCA beschrieben und die Probleme aufgezeigt, die sich aus dem aktuellen Lokalisierungsprozess ergeben.

5.1 Der Aufbau von IUCCA

IUCCA ist eine webbasierte Anwendung, auf welche die Benutzer mittels eines Browers über das Internet bzw. das Intranet von DaimlerChrysler zugreifen können. Also arbeiten alle Mitarbeiter der einzelnen TruckStores in den unterschiedlichen Ländern auf einem zentralen Server. Das ist von großem Vorteil für die Pflege und die Administration von IUCCA.

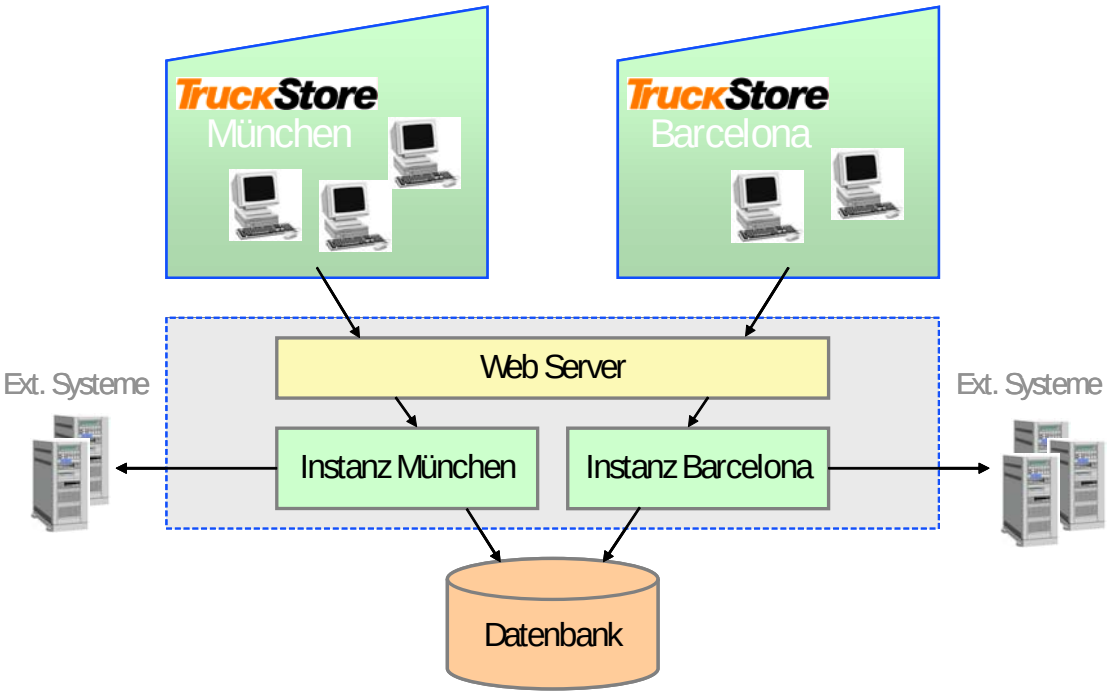

Abbildung 4: IUCCA-Instanzen [Leutner05]

Aus Abbildung 4 wird ersichtlich, dass jeder TruckStore eine eigene IUCCA-Instanz besitzt. Alle TruckStores greifen auf den gleichen Webserver zu und werden von diesem an die entsprechende IUCCA-Instanz weiter geleitet. Der parallele Betrieb von mehreren Instanzen gewährleistet bei Auftreten eines Problems mit einer Instanz, dass

die anderen TruckStores auf ihren Instanzen problemlos weiter arbeiten können. Viel wichtiger ist aber, dass durch die Instanzen die Mandantenfähigkeit von IUCCA realisiert wird. So greift die Instanz München auf andere externe Systeme zu, als die Instanz des TruckStores in Barcelona. In der spanischen Instanz hingegen wird ein anderes Modul zur Steuerberechung verwendet als in der deutschen Instanz.

Um die Mandantenfähigkeit umzusetzen, wurde bei der Architektur des Systems darauf geachtet, alle `Locale`-abhängigen Bestandteile aus dem Systemkern herauszulösen und als so genannte Länder- bzw. Sprachpakete zu kapseln.

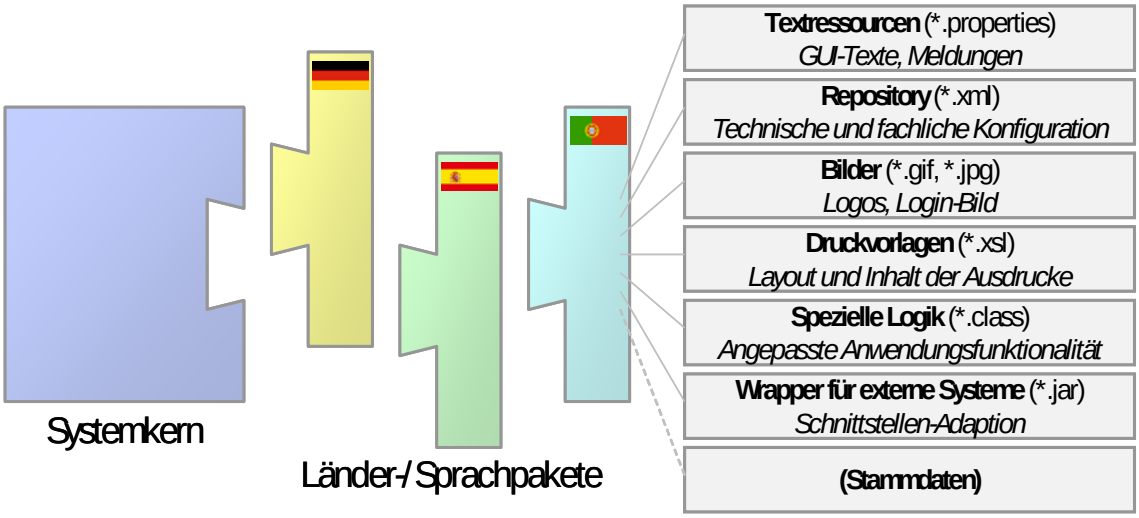

Abbildung 5: Systemkern und Länderpakete von IUCCA [Leutner05]

Die Abbildung 5 zeigt schematisch die Aufteilung von IUCCA in den Systemkern und die einzelnen Länderpakete. Ein Länderpaket enthält alle landesspezifischen Ressourcen für ein bestimmtes Land. Wenn für einen bestimmten TruckStore eine neue Instanz gebaut werden soll, muss das entsprechende Länderpaket gemeinsam mit dem Systemkern installiert werden.

5.2 Internationalisierungstechniken

Für die Internationalisierung des IUCCA Programmcodes wurden die in Kapitel 4.4 beschriebenen Java-Technologien verwendet. In diesem Abschnitt soll nun das Laden von Ressourcen noch einmal betrachtet werden. IUCCA verwendet zum Laden von Sprachressourcen das `PropertiesResourceBundle`. Dessen Funktionalität wurde jedoch erweitert. In IUCCA sind in den Properties-Dateien neben den herkömmlichen Schlüssel-/Wert-Paaren auch Schlüssel erlaubt, deren Werte Verweise auf andere Schlüssel enthalten. Die Verweise haben die Form `${schlüsselname}`. Das könnte in der Datei »text.properties« wie folgt aussehen:

```
#Herkömmliche Schlüssel- /Wert- Paare
baseCommission=Regelprovision
dealer=Händeler
#Schlüssel mit Verweisen auf andere Schlüssel.
baseCommissionDealer=${baseCommission} ${dealer}
```

Darüber hinaus sind Verweise zwischen Properties-Dateien erlaubt. So kann aus der Datei »messages.properties« auf den Schlüssel der Datei »text.properties« verwiesen werden. Wobei die Datei »text.properties« als Default-Datei für Verweise angesehen wird. So referenziert ein Verweis der Form `${schlüsselname}` immer auf »text.properties«.

Soll auf eine andere Properties-Datei referenziert werden, muss im Verweis der Name der zu referenzierenden Datei angegeben werden. Dabei wird die Endung .properties weggelassen. Ein solcher Verweis hat die Form `${dateiname:schlüsselname}`. So könnten in »messages.properties« folgende Schlüssel stehen:

```
#Herkömmlicher Schlüssel
missingEntry=Angabe fehlt.
#Verweis auf einen Schlüssel in »text.properties« und einen
#Schlüssel in »messages.properties«
reservation.salespersonNotSet=${dealer}: ${messages:missingEntry}
```

Diese Erweiterung des `PropertiesResourceBundle` hat den Sinn, dass bestimmte Begriffe nur einmal übersetzt werden müssen und in allen Sprachressourcen gleich verwendet werden. Allerdings hat diese Erweiterung zur Folge, dass für Übersetzer ohne technischen Hintergrund das Bearbeiten der Properties-Dateien schwieriger wird.

Für die Generierung der dynamischen HTML-Benutzeroberflächen wird in IUCCA das OSE[16]-Framework verwendet. OSE ist ein Web-Framework, welches in Aufbau und Funktionalität dem in Kapitel 7.4 beschriebenen Struts-Framework gleicht. Wie Struts bietet auch OSE zusätzlich zu den HTML-Tags eigene, sogenannte *Custom Tags* an. In diesen *Custom Tags* werden Funktionalitäten des Frameworks gekapselt. Diese können dann über die Auszeichnungssprache XML[17] in den HTML-Code eingebunden werden.

Die Internationalisierung der Oberflächen unterstützt OSE durch das *Custom Tag* `<ose:text key="schlüssel">`. Dieses *Tag* greift mit Hilfe des `PropertyResourceBundle` auf die, dem *Locale* des Benutzers entsprechenden Properties zu und stellt den Wert des Schlüssels in der Sprache des Anwenders dar.

[16] OSE – Open Servlet Environment

[17] XML - Extensible Markup Language. Siehe auch: www.w3.org/XML/

5.3 Entwicklungsprozess

Der Softwareentwicklungsprozess von IUCCA richtet sich nach Vorgehensmodell für *eXtreme Programming* (XP). *Extreme Programming* ist eine relativ neue Vorgehensweise in der Softwaretechnik. Sie ist ein Prozess aus der Klasse der agilen Prozesse der Softwareentwicklung.

5.3.1 Wasserfallmodell

Ein nichtagiler Prozess wäre z.B. das klassische Wasserfallmodell. Das Wasserfallmodell bezeichnet das traditionelle Vorgehensmodell der Softwareentwicklung. Im Wasserfallmodell werden mehrere Phasen definiert und jede Phase hat definierte Start- und Endpunkte mit eindeutig definierten Ergebnissen. Ist eine Phase abgeschlossen, so kann man nicht mehr zu dieser Phase zurückkehren. Dies kann man mit den Kaskaden eines Wasserfalls vergleichen, woher das Modell auch seinen Namen hat. [WikiWF05] benennt folgende Phasen:

- Anforderungsanalyse und -spezifikation (engl. Requirement analysis and specification)
- Systemdesign und -spezifikation (engl. *System design and specification*)
- Programmierung und Modultests (engl. *Coding and module testing*)
- Integration und Systemtests (engl. *Integration and system testing*)
- Auslieferung, Einsatz und Wartung (engl. *Delivery and maintenance*)

Eine andere Variante nach [WikiWF05] macht daraus sechs Schritte:

- Planung (engl. System Engineering)
- Definition (engl. Analysis)
- Entwurf (engl. Design)
- Implementierung (engl. Coding)
- Testen (engl. Testing)
- Einsatz und Wartung (engl. Maintenance)

5.3.2 eXtreme Programming (XP)

In XP wird auf einen strikten Anforderungskatalog des Kunden verzichtet. Dafür werden auch Kundenwünsche berücksichtigt, die sich noch während der Softwareentwicklung ergeben. In der Praxis hat sich ergeben, dass dem Kunden zu Projektbeginn grundlegende Anforderungen nur unzulänglich bekannt sind. Er fordert Funktionalitäten, die er nicht braucht und vergisst solche, die benötigt werden.

Statt des klassischen Wasserfallmodells durchläuft der Entwicklungsprozess immer wieder und in kurzen Zyklen sämtliche Disziplinen der klassischen Softwareentwick-

lung, also Anforderungsanalyse, Design, Implementierung und Test. Nur die im aktuellen Iterationsschritt benötigten Anforderungen werden implementiert. In IUCCA beträgt ein solcher Zyklus zwei Wochen.

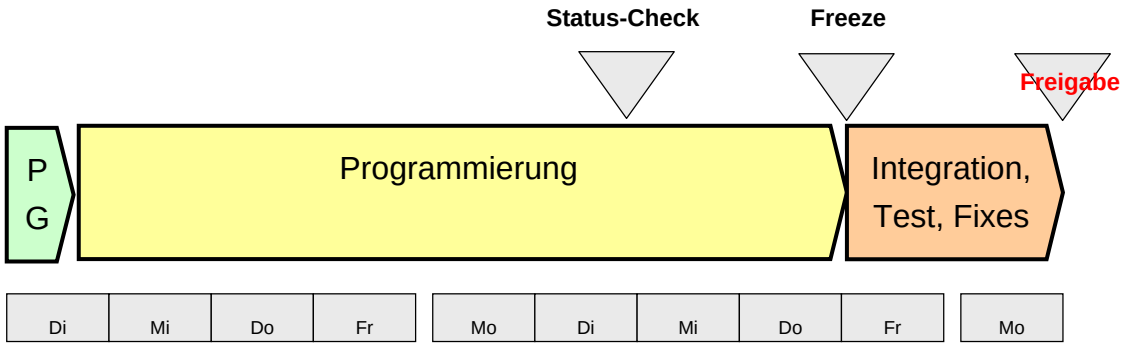

Abbildung 6: XP Zyklus im IUCCA Projekt

In Abbildung 6 wird ein typischer Zyklus nach dem XP-Vorgehensmodell gezeigt. Der Zyklus beginnt mit dem sogenannten Planning Game (PG) (dt. Planspiel). Im Planning Game wird der letzte Zyklus analysiert. Anschließend setzt die Analyse der neuen Anforderungen ein. Danach werden die Aufgaben an die einzelnen Entwickler verteilt, die sogleich eine Aufwandsschätzung machen. Nach dem Planning Game kann mit der Programmierung der neuen Aufgaben begonnen werden. Nach einer Woche findet ein Statusmeeting statt, in dem der aktuelle Status des Zyklus ermittelt wird. Bei auftretenden Problemen können entsprechende Maßnahmen eingeleitet werden. Wenn der Zeitpunkt des so genannten *Freeze* (dt. einfrieren) kommt, werden alle Programmierarbeiten eingestellt. Die aktuelle Version der Anwendung wird auf einem speziellen Testserver gebaut, der nur für interne Tests vorgesehen ist. Nachdem getestet und eventuelle Fehler behoben wurden, wird die Version auf einem Integrationsserver gebaut. Auf diesem Server können nun auch die Kunden die Anwendung testen. Sobald der Kunde mit dem Ergebnis zufrieden ist, gibt er die Freigabe für diese Version. Jetzt kann die aktuelle Version auch auf dem Produktionsserver installiert werden. Sollte der Kunde Reklamationen haben, so werden diese als Aufgaben in den nächsten Zyklus übernommen.

Diese kurzen Zyklen ermöglichen es dem Kunden das Projekt sehr genau zu steuern und zu kontrollieren. Das Entwicklungsteam hat im XP-Modell mehr Freiheiten aber auch mehr Verantwortung als in den klassischen Modellen. Das IUCCA-Team konnte mit Hilfe dieses agilen Vorgehensmodells sowohl hohe Produktivität und Qualität als auch hohe Kundenzufriedenheit erzielen.

Allerdings kommt es durch die kurzen Iterationsphasen zu Problemen bei der Lokalisierung von Sprachressourcen. Diese werden deutlich, wenn man den Lokalisierungsprozess für die Sprachressourcen genauer betrachtet.

5.4 Lokalisierungsprozess der Sprachressourcen

In diesem Kapitel wird der Lokalisierungsprozess der Sprachressourcen von IUCCA genauer erläutert. Zwar muss bei der Lokalisierung von IUCCA auch beachtet werden, dass für die einzelnen Länder unterschiedliche Module, z.B. zur Berechnung von Steuern oder Provisionen, existieren. Diese Problematik wird aber nicht im Rahmen dieser Diplomarbeit betrachtet. Für nähere Informationen zu diesem Thema möchte ich auf die Arbeit von [Krämer04] verweisen.

Wie bereits erwähnt, kommt es durch die kurzen Iterationszyklen in der IUCCA-Entwicklung zu Problemen mit der Lokalisierung der Sprachressourcen. Der bisherige Lokalisierungsprozess in IUCCA sieht folgendermaßen aus:

- Die deutschen Properties-Dateien sind die Referenz für alle lokalisierten Properties. Das bedeutet, dass alle Änderungen nur in den deutschen Properties-Dateien vorgenommen werden.
- Die Entwickler fügen den deutschen Properties-Dateien neue Texte hinzu, die später übersetzt werden müssen.
- Nach der Bearbeitung der Properties-Dateien werden diese in das Concurrent Versions System[18] (CVS) des Entwicklungsteams eingepflegt.
- Für die Vorbereitung des Übersetzungspakets werden nun sowohl die deutschen als auch die bereits lokalisierten Properties-Dateien aus dem CVS ausgecheckt. Ein zu diesem Zweck geschriebenes Programm vergleicht nun die Properties und fügt die neuen Werte den lokalisierten Properties hinzu und markiert diese (siehe auch [Krämer04]). So stehen in den z.B. spanischen Properties deutsche Texte.
- Anschließend werden die lokalisierten Properties-Dateien in das Comma Separated Value[19] (CSV) Dateiformat umgewandelt. Diese CSV-Dateien werden nun in das Tabellenkalkulationsprogramm Excel der Firma Microsoft importiert.
- Die so erzeugten Excel-Dateien werden dann an den zuständigen Übersetzer des jeweiligen Landes geschickt.
- Der Übersetzer kann daraufhin die neu hinzugekommenen Texte in MS Excel bearbeiten. Allerdings kann er nicht erkennen, in welchem Kontext die Texte später in der Anwendung auftreten. Daher kann es leicht zu nicht sinngemäßen Übersetzungen kommen.

[18] CVS ist eine Anwendung, die das Verwalten von Änderungen im Programmcode unterstützt.

[19] CSV ist ein Dateiformat, bei dem die einzelnen Felder durch Kommas getrennt sind. Dieses Format wird oft für die Übertragung von Daten zwischen unterschiedlichen Systemen genutzt.

- Nach der Bearbeitung sendet der Übersetzer die Excel-Dateien wieder an die Entwickler, diese müssen die Excel-Dateien in Java Properties-Dateien umwandeln und wieder ins CVS einpflegen.
- Die übersetzten Properties-Dateien werden beim Bauen der nächsten IUCCA-Version integriert und erst dann sind die Änderungen sichtbar.
- Sollte der Kunde beim Testen dieser neuen Version einen Fehler in der Übersetzung finden, muss er Kontakt mit dem zuständigen Entwickler aufnehmen und ihm entweder die gewünschte Änderung mitteilen oder sich die Sprachressourcen erneut zuschicken lassen, um die Änderungen selbst durchzuführen. Dafür muss der gesamte Prozess erneut durchlaufen werden.

Am dem beschriebenen Prozess kann man nun mehrere Probleme feststellen. So ist der Prozess an sich recht umständlich. Er hat einen hohen Kommunikationsbedarf zwischen Entwicklern und Kunden und auch innerhalb des Entwicklerteams zur Folge.

Auch das Umwandeln der Dateien ist recht fehleranfällig. Vor allem, wenn mit Griechenland und Russland neue Schriftsätze hinzukommen. Wie in Kapitel 4.4.2.5 beschrieben, müssen Java Properties-Dateien ISO-8859-1 kodiert sein. Russisch und Griechisch sind hingegen in ISO-8859-5 bzw. in ISO-8859-7 kodiert. Eine zusätzliche Umwandlung zwischen den Zeichensätzen würde die Fehleranfälligkeit des Prozess deutlich erhöhen.

Die für die Übersetzung zuständigen Mitarbeiter des Kunden führen die Übersetzungen zusätzlich zu ihrem normalen Arbeitspensum durch. Daher kann das Bearbeiten der Sprachressourcen oft viel Zeit in Anspruch nehmen. Die Tatsache, dass die Übersetzer den Kontext der Texte nicht kennen und das Ergebnis ihrer Arbeit oft erst einige Wochen später sehen können, steigert nicht deren Motivation die Texte zügig zu bearbeiten.

Die Lokalisierung der Textressourcen kann also oft einige Zeit in Anspruch nehmen. Durch den XP-Ansatz sind jedoch kurze Iterationszyklen vorgegeben. Wenn nun die lokalisierten Properties noch nicht zur Verfügung stehen, kann es vorkommen, dass auf dem Integrationsserver eine Version gebaut werden muss, die noch nicht übersetzte Texte enthält.

Dies ist die Ausgangssituation für diese Diplomarbeit. Im Zuge dieser Arbeit sollte ein neuer, schlanker Prozess entwickelt werden, der den alten ersetzen soll. Dieser Prozess soll durch eine webbasierte Anwendung abgebildet werden, die möglichst viele Schritte automatisiert und den Lokalisierungsprozess in die Iterationszyklen des XP-Vorgehensmodells einbindet.

6 Anforderungsanalyse und Planung von TROIA

In diesem Kaptitel wird gezeigt, wie bei dem Entwurf der Architektur für TROIA vorgegangen wurde. Dafür werden die einzelnen Schritte und Überlegungen, die beim Entwurf eine Rolle gespielt haben genauer beschreiben.

6.1 Vorgegebene Anforderungen

Der erste Schritt bei dem Entwurf einer Architektur für TROIA, war die Erfassung und Aufstellung der vorgegebenen und bisher bekannten Anforderungen an TROIA.

- TROIA soll den neuen Lokalisierungsprozess mit den XP-Iterationszyklen synchronisieren.
- Die Sprachressourcen sollen gleichzeitig von mehreren Übersetzern, aber auch von den Entwicklern des IUCCA-Teams bearbeitet werden können.
- Es soll auf einem zentralen System mit gemeinsamer Datenbasis gearbeitet werden. Das Verschicken von Dateien soll nicht mehr nötig sein.
- Die Anwendung TROIA soll aus der Oberfläche der IUCCA-Anwendung heraus aufgerufen werden können, um dadurch das komfortable Korrigieren von Fehlern zu erlauben.
- Die Übersetzer sollen die Möglichkeit haben, einen bestimmten Begriff im Kontext der Anwendung zu erkennen. Darüber hinaus sollen die Anwender ihre Änderungen sofort auf der Oberfläche von IUCCA sehen.
- Die Anwendung TROIA soll eine intuitive Oberfläche bieten und alle für das effektive Bearbeiten der Sprachressourcen benötigten Funktionen zur Verfügung stellen.
- TROIA soll die Versionisierung der Sprachressourcen unterstützen.
- Für die Realisierung von TROIA soll die Programmiersprache Java eingesetzt werden. Außerdem sollen *Open Source* Produkte verwendet werden.

6.2 Evaluierung verschiedener Anforderungen an das Gesamtsystem

Nachdem die grundlegenden Anforderungen an TROIA feststanden, mussten einige grundsätzliche Entscheidungen über die zukünftige Verwendung von Sprachressourcen in IUCCA und über das Zusammenspiel von TROIA und IUCCA getroffen werden. Aus diesen Entscheidungen ergaben sich weiter Anforderungen an TROIA. Erst danach konnte der neue Lokalisierungsprozess definiert und eine detaillierte Architektur entworfen werden.

6.2.1 Zugriff auf Sprachressourcen

Die erste Überlegung war, ob die Sprachressourcen für IUCCA auch zukünftig in Java-Properties-Dateien gespeichert werden. Eine gute Alternative zu diesem Ansatz wäre, die Sprachressourcen in einer Datenbank zu verwalten. Dieser Ansatz hätte den Vorteil, dass man sehr viel generischer arbeiten könnte, da alle manuellen Prozesse bei der Verwaltung der Java Properties-Dateien wegfallen. Die Daten könnten dann mit TROIA bequem bearbeitet werden.

Technisch kann dies durch die Erweiterung der verwendeten `ResourceBundle`-Klasse geschehen. Das `ResourceBundle` würde dann die Sprachressourcen statt aus den Properties-Dateien, aus einer Datenbank lesen und der Anwendung zur Verfügung stellen. Allerdings zeigte sich, dass die Erweiterung des Properties-Mechanismus um die in Kapitel 5.2 beschriebenen Verweise erheblich mehr Änderungen im IUCCA Programmcode zur Folge hätten, als ursprünglich angenommen.

Da jedoch aus wirtschaftlichen Gründen die Änderungen im IUCCA Programmcode möglichst gering gehalten werden sollen, wurde beschlossen, in IUCCA das Konzept der Java Properties-Dateien beizubehalten. In TROIA soll jedoch eine Datenbank für die Verwaltung der IUCCA-Sprachressourcen verwendet werden. Also muss TROIA auch Schnittstellen für das Einlesen der Properties-Dateien in die Datenbank, bzw. für das Erzeugen von Properties aus der Datenbank zur Verfügung stellen.

6.2.2 Zugriff von IUCCA auf TROIA

Nachdem die Frage der Sprachressourcen geklärt war, musste überlegt werden, wie aus IUCCA heraus auf TROIA zugegriffen werden soll. Dabei stand von Anfang an fest, dass die Kommunikation zwischen IUCCA und TROIA ausschließlich über das HTTP[20]-Protokoll erfolgen muss. So musste noch festgelegt werden, wie der Aufruf von TROIA in IUCCA stattfinden soll. Dafür gab es mehrere vorstellbare Szenarien:

[20] HTTP - Hypertext Transfer Protocol, ist ein zustandsloses Datenaustausch-Protokoll zur Übertragung von Daten. Primär wird es im Rahmen des World Wide Web zur Übertragung von Webseiten verwendet.

- Szenario 1: Anwender mit bestimmten Rechten können jeden beliebigen Text an der Oberfläche anklicken und daraufhin von TROIA die Möglichkeit bekommen, diesen Text zu editieren.
- Szenario 2: TROIA kann über einen Link im Menü von IUCCA aufgerufen werden. Dem Anwender wird eine Suchmaske zum Finden des gewünschten Begriffs angeboten.
- Szenario 3: TROIA wird über einen Link im Menü von IUCCA aufgerufen. Jedoch zeigt TROIA dem Anwender gleich alle Texte der aktuellen IUCCA-Seite zum Bearbeiten an. Darüber hinaus hat der Anwender aber auch die Möglichkeit Texte zu suchen.

Bei der Beurteilung der einzelnen Szenarien stellte sich heraus, dass das erste Szenario zwar die für den Benutzer komfortabelste Lösung bietet, aber auch wieder erhebliche Anpassungen im IUCCA Programmcode nach sich zieht. Für Szenario 2 dagegen sind die Änderungen im Code minimal. Es bedarf nur eines statischen Links auf TROIA. Allerdings ist es für den Benutzer unbequem, wenn er einen Begriff erst über eine Suchmaske suchen muss. Also hat man sich für das dritte Szenario entschieden.

Auch in Szenario 3 sind die Änderungen im IUCCA-Programmcode minimal. Dem Link müssen lediglich einige Parameter über die aktuelle Seite mitgegeben werden. Allerdings hat der Anwender gleich eine Liste aller sichtbaren Texte zur Verfügung und kann so alle Texte der aktuellen Seite bequem bearbeiten.

Für TROIA bedeutet die Wahl des dritten Szenarios, dass nicht nur die Sprachressourcen in der Datenbank gehalten werden müssen, sondern zusätzlich auch Informationen über die Verwendung der einzelnen Ressourcen in den IUCCA-Seiten.

6.2.3 Bearbeitung von Sprachressourcen

Da nun die grundlegenden technischen Rahmenbedingungen festgelegt waren, musste noch geklärt werden, ob die Properties-Dateien in Zukunft ausschließlich mit TROIA bearbeitet werden dürfen oder ob auch weiterhin eine manuelle Bearbeitung außerhalb von TROIA erlaubt ist.

Für die Übersetzer stellt TROIA eine komfortable Möglichkeit für das Bearbeiten von Sprachressourcen dar. Die Entwickler von IUCCA empfanden es allerdings als umständlich, jedes Mal eine Anwendung außerhalb ihrer Entwicklungsumgebung starten zu müssen. Um den Bedenken der Entwickler entgegenzukommen, wurde festgelegt, dass die Default-Properties auch weiterhin von den Entwicklern manuell bearbeitet werden dürfen. Die lokalisierten Properties-Dateien werden hingegen in Zukunft nur noch in TROIA bearbeitet. Das bedeutet, dass TROIA beim Laden der Sprachdaten in die Datenbank, die Veränderungen in den Default-Properties auf die lokalisierten Properties abbilden muss.

6.3 Der Use Case von TROIA

Auf der Grundlage der bisher getroffenen Entscheidungen und den bereits bekannten Anforderungen konnte nun ein UML[21]- *Use Case*-Diagramm für TROIA erstellt werden.

Ein *Use Case* beschreibt die Art und Weise, wie ein Akteur auf ein System einwirkt. Er ist eine Beschreibung für das äußerlich sichtbare Verhalten des Systems. *Use Cases* sind ein Hilfsmittel zur Anforderungsanalyse. Das heißt, sie sollen verdeutlichen was das System leisten soll, aber nicht wie das System etwas leisten soll.

Die Aufgabe des in Abbildung 7 dargestellten Use Case-Diagramms, ist also die Visualisierung des Funktionsumfangs von TROIA auf der Grundlage der Anforderungen.

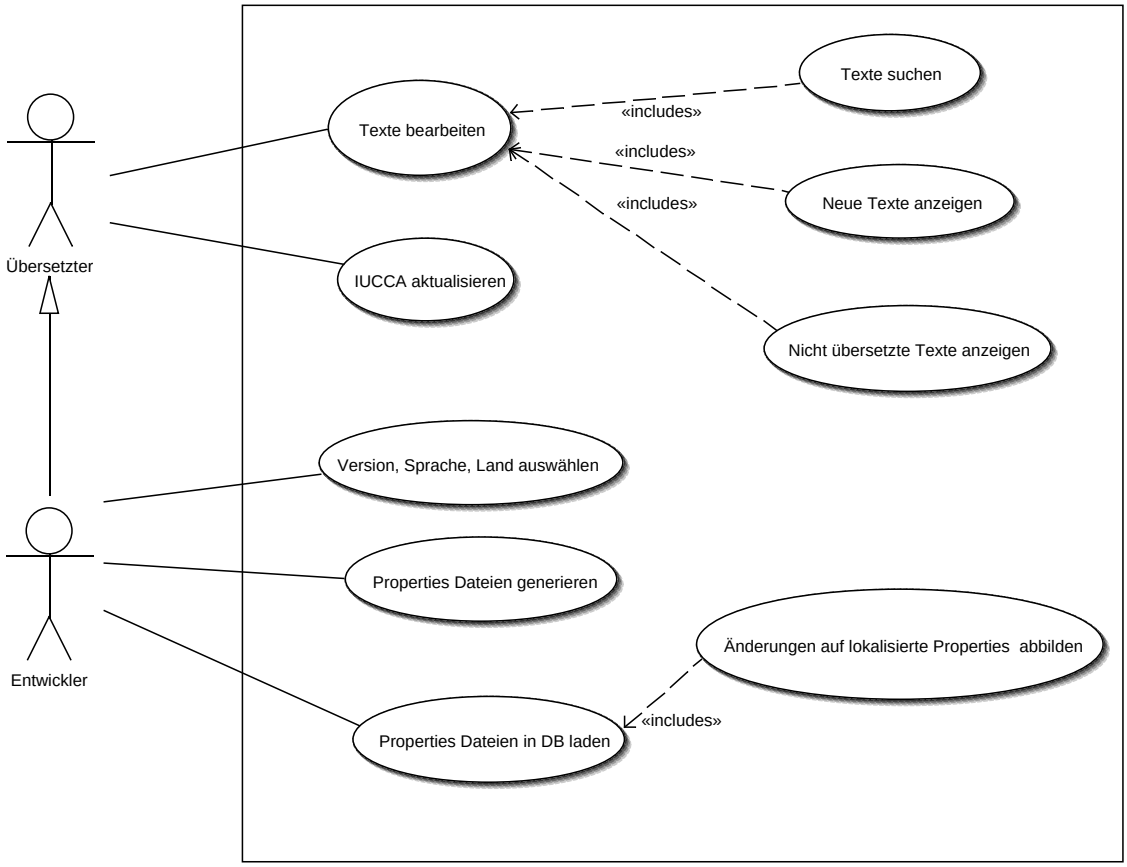

Abbildung 7: UML-Use-Case-Diagramm für TROIA

In dem Use Case-Diagramm für TROIA werden die Anwender (Akteure) und die ihnen zur Verfügung stehenden Funktionalitäten dargestellt.

Die Übersetzer können in TROIA

- Texte bearbeiten, was die Suche nach bestimmten Texten, sowie das Anzeigen von neuen bzw. nicht übersetzten Texten beinhaltet. Dabei sind sie auf Texte

[21] UML – Unified Modeling Language. Siehe auch [UML05]

beschränkt, die durch die Sprache und Version der von ihnen benutzten IUCCA-Instanz vorgegeben werden.

- die Aktualisierung von IUCCA bewirken und sich dadurch die geänderten Texte anzeigen lassen.

Den Entwicklern steht der gesamte Funktionsumfang der Übersetzer zur Verfügung. Darüber hinaus bietet TROIA ihnen die Möglichkeit:

- die Texte aller Sprachen, Länder und Version zu bearbeiten.
- die Daten von Properties-Dateien in die Datenbank von TROIA zu laden.
- neue Properties zu generieren.

6.4 Der neue Lokalisierungsprozess aus Anwendersicht

Parallel zum *Use Case*-Diagramm wurde auch der neue Lokalisierungsprozess entworfen, da der Prozess Einfluss auf den Funktionsumfang von TROIA haben kann. Umgekehrt hat der Funktionsumfang von TROIA auch Einfluss auf den Prozess. Da das *Use Case*-Diagramm das System aus Anwendersicht darstellt, wurde auch der Prozess zuerst aus Anwendersicht definiert.

Der Ablauf des neuen Prozesses sieht folgendermaßen aus:

- Unmittelbar nachdem die Programmierung in einem Zyklus abgeschlossen ist und eine neue Version auf dem Test- bzw. Integrationsserver gebaut wird, wird die TROIA-Funktion zum Laden der Sprachressourcen in die Datenbank gestartet.
- Nun kann der Übersetzer die neue IUCCA-Version testen. Sollte er dabei nicht übersetzte Texte finden, kann er TROIA direkt aus IUCCA heraus aufrufen. In TROIA werden ihm alle Texte der aktuellen Seite angezeigt und er kann den gewünschten Text bearbeiten.
- Nach dem Bearbeiten der Texte kann der Übersetzer IUCCA aktualisieren und sich dadurch die Änderungen gleich an der Oberfläche anzeigen lassen. Dadurch kann er auch gleich erkennen, ob der neue Text eine zu lange Lauflänge hat und dadurch das Layout der entsprechenden Seite verändert.
- Nach dem Aufruf von TROIA kann der Übersetzer sowohl die auf den Seiten angezeigten Texte als auch Texte von Meldungen oder Menüs, die keiner speziellen Seite zugeordnet sind, bearbeiten. Auch hat er die Möglichkeit, sich alle neu hinzugekommenen Texte sowie noch nicht übersetzte Texte anzeigen zu lassen und diese zu bearbeiten.
- Anwender, die TROIA aus IUCCA heraus aufrufen, können nur Texte für die aktuelle Version ihres Landes anzeigen und bearbeiten.

- Entwickler bzw. Administratoren haben direkten Zugang zu TROIA und können dort die Texte für alle bisher erfassten Versionen, Länder und Sprachen bearbeiten.
- Sobald die Entwickler den aktuellsten Stand einer lokalisierten Properties-Datei benötigen, z.B. wenn eine neue Version gebaut werden soll, lassen sie sich von TROIA die gewünschten Properties-Dateien erzeugen und pflegen diese in das CVS ein.
- Anschließend kann eine neue Version gebaut werden. Sollte es in den Default-Properties zu Änderungen kommen, werden die Daten wieder nach TROIA zur Bearbeitung übertragen.

Der neue Lokalisierungsprozess ist nicht mehr auf die Umwandlung und das Verschicken der Properties-Dateien angewiesen, da die Daten nun zentral gespeichert sind und über eine webbasierte Oberfläche allen berechtigten Benutzern zur Verfügung stehen. Das entlastet die Entwickler und reduziert den Kommunikationsaufwand.

Da TROIA Unicode verwendet und dem Java-Standard entsprechende Properties-Dateien erzeugt, können zum Bearbeiten der Texte beliebige Schriftzeichen verwendet werden. Dies verhindert Kodierungsprobleme, die durch griechische und kyrillische Zeichensätze unweigerlich aufgetreten wären. Wollte man dieses Problem im herkömmlichen Prozess in den Griff bekommen, müsste man sicherstellen, dass alle Beteiligten in ihren Textbearbeitungsprogrammen den gleichen Zeichensatz, z.B. UTF-8, verwenden. In der Praxis wäre das nicht leicht umzusetzen.

Durch die zentrale Datenhaltung können alle Übersetzer parallel auf den lokalisierten Texten arbeiten und die Entwickler haben jederzeit Zugriff auf die aktuellste Version der lokalisierten Properties. Dadurch passt sich der neue Lokalisierungsprozess sehr viel besser in den agilen Ansatz des XP-Vorgehensmodells ein als der herkömmliche Prozess.

6.5 Der neue Lokalisierungsprozess aus technischer Sicht

Nach dem nun der neue Lokalisierungsprozess sowie der grobe Funktionsumfang von TROIA festgelegt wurde, konnte man sich nun Gedanken über die technische Umsetzung von TROIA machen.

Der Entwurf einer möglichen Architektur für TROIA basierte auf der Analyse der geforderten Funktionen. Dabei wurde festgestellt, dass die Funktion des Ladens der Sprachressourcen in die Datenbank von TROIA einen entscheidenden Einfluss auf die Architektur des Gesamtsystems haben könnte.

Das Problem bestand darin, dass festgelegt werden musste, wie der Prozess des Ladens der Daten gestartet werden sollte. Die eine Möglichkeit wäre, dass ein Entwickler TROIA startet nachdem er eine neue Version gebaut hat. Über TROIA baut er dann

eine Verbindung zum CVS auf und gibt TROIA die benötigten Ressourcen an. Dabei sind zwei Aspekte zu beachten:

1. TROIA benötigt nicht, wie in Kapitel 6.2.2 beschrieben, nur die Sprachressourcen sondern auch Informationen, wo die Sprachressourcen an der Oberfläche von IUCCA verwendet werden.
2. Die Verzeichnisstruktur, in welcher der Programmcode von IUCCA im CVS verwaltet wird, ist recht komplex, da es globale und länderspezifische Dateien gibt. Das bauen einer neuen IUCCA-Instanz ist also auch ein komplexer Prozess, der mit Hilfe der in Kapitel 7.5 beschriebenen Ant-Build-Scripte erfolgt. In diesen Scripten werden die benötigten Ressourcen zusammengetragen und anschließend zu einer Anwendung kompiliert.

Daraus folgt, dass der Entwickler beim manuellen Starten der Funktion viele Daten übergeben muss, die er bereits beim Anpassen des Build-Scripts bearbeitet hat. Es wäre daher geschickt, wenn die Ladefunktion direkt im Build-Script gestartet werden könnte.

Des Weiteren müssen die Daten der Sprachressourcen aufbereitet und mit zusätzlichen Informationen versehen werden, bevor diese in die Datenbank gespeichert werden können. Aus diesem Grund wurde beschlossen, dass man das Laden und Verarbeiten der Daten in ein eigenes Programm, den TROIA-Parser, auslagert, das sich als JAR[22]-Datei auf dem jeweiligen Server befindet und von Ant in Form eines Ant-Task[23] aufgerufen werden kann.

TROIA besteht also aus zwei unabhängigen Anwendungen. Zum einen gibt es die webbasierte Anwendung TROIA, in der die in der Datenbank gespeicherten Daten der Properties-Dateien bearbeitet werden können. Zum anderen gibt es den TROIA-Parser, der die Properties-Dateien und die Java Server Pages von IUCCA analysiert und die erzeugten Daten in die Datenbank speichert.

[22] JAR – Java Archiv. Ein JAR ist eine komprimierte Datei zur Verteilung von Java Klassen.
[23] siehe Kapitel 8.1.1

Abbildung 8 zeigt den schematischen Aufbau des Gesamtsystems.

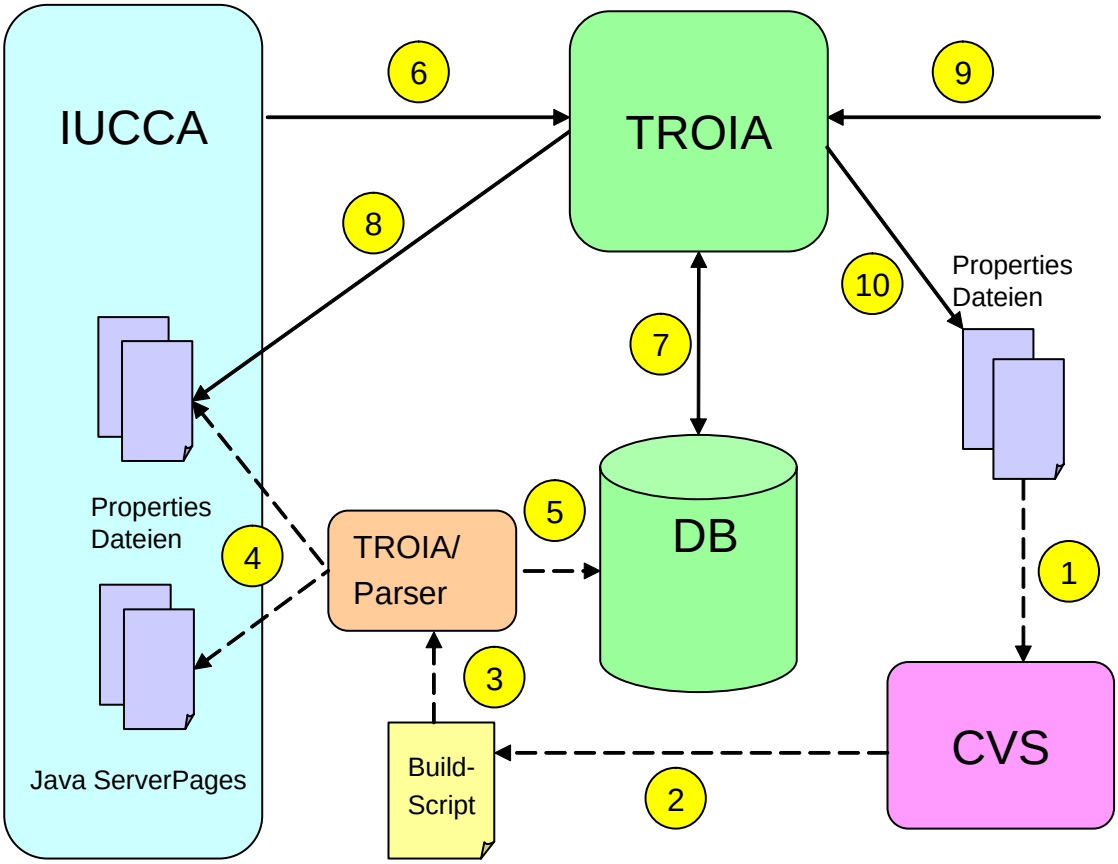

Abbildung 8: Übersicht IUCCA und TROIA

Nachdem die generelle Architektur des Gesamtsystems festgelegt wurde, kann der Prozess nun auch aus technischer Sicht betrachtet werden:

1. Aktuelle Java Properties-Dateien werden in das CVS eingepflegt.
2. Durch ein Ant-Build-Script wird eine IUCCA Instanz gebaut.
3. Der TROIA-Parser wird aus dem Build-Script heraus gestartet. Dabei werden ihm Informationen über die Version, das *Locale* und der Verzeichnispfad der Instanz übergeben.
4. Der TROIA-Parser analysiert die JSP-Dateien und merkt sich, welche Schlüssel in welcher Seite verwendet werden. Anschließend werden die Differenzen zwischen den Default- und den lokalisierten Ressourcen ermittelt. Neue Texte werden den lokalisierten Sprachressourcen hinzugefügt und Texte, die in den Default-Ressourcen weggefallen sind werden auch aus den lokalisierten Sprachressourcen gelöscht.
5. Der TROIA-Parser schreibt die erzeugten Daten in die Datenbank.

6. Ein Übersetzer ruft TROIA aus IUCCA heraus auf. Dabei werden TROIA Informationen über die Version, das *Locale*, die aktuelle Seite und die URL der Instanz übergeben.
7. Aufgrund der erhaltenen Informationen lädt TROIA die benötigten Daten aus der Datenbank und schreibt die bearbeiteten Daten wieder in die Datenbank zurück.
8. Wenn der Übersetzer die Funktion »IUCCA aktualisieren« aufruft, erzeugt TROIA die neuen Properties-Dateien und übermittelt diese per http an IUCCA. Daraufhin aktualisiert IUCCA seine Properties-Dateien und stellt die Änderungen dar.
9. Entwickler können TROIA direkt aufrufen und die Version und das Locale der gewünschten Sprachressourcen frei wählen. Anschließend können die Daten bearbeitet werden.
10. Sollten die Entwickler die aktuellste Version der Properties-Dateien benötigen, z.B. um eine neue IUCCA-Instanz zu bauen, können sie sich von TROIA die entsprechenden Dateien erzeugen lassen, und diese an einem beliebigen Ort speichern.

6.6 Verteilungsdiagramm des Gesamtsystems

Im nächsten Schritt in der Planung von TROIA musste ermittelt werden, auf welche Knoten die einzelnen Komponenten verteilt sind. Als Knoten sind in diesem Zusammenhang physische Objekte, also die eingesetzten Server zu verstehen.

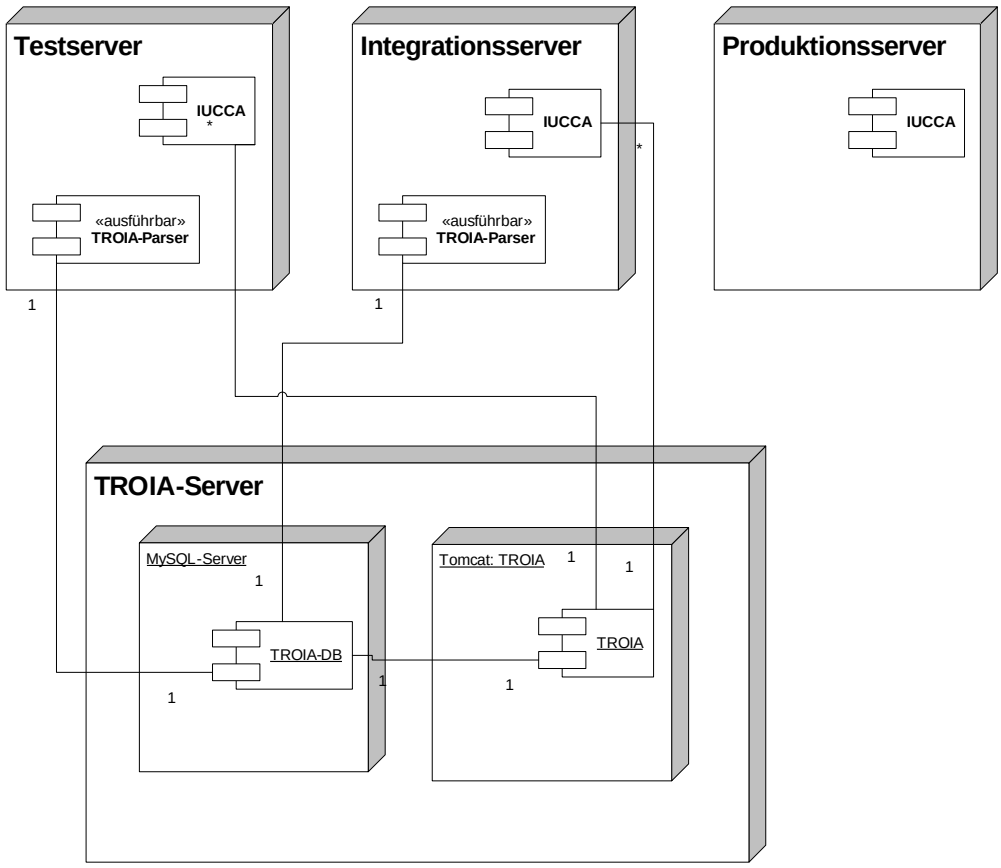

Abbildung 9: Deployment-Diagramm für das Gesamtsystem

Abbildung 9 zeigt die Verteilung des Gesamtsystems. Man sieht, dass das TROIA – System auf einem eigenen Server läuft. Auf diesem Server befindet sich sowohl die TROIA-Datenbank, als auch die Anwendung selbst. Weiterhin erkennt man, dass sich sowohl auf dem Test- als auch auf dem Integrationsserver je eine Instanz des TROIA-Parser befindet, die bei Bedarf gestartet werden kann und Daten an die TROIA-Datenbank übermittelt.

TROIA kann nur von IUCCA-Instanzen angesprochen werden, die sich entweder auf dem Test- oder Integrationsserver befinden. Es ist nicht geplant, TROIA den IUCCA-Instanzen auf dem Produktionsserver zur Verfügung zu stellen.

7 Verwendete Technologien

Im letzten Schritt der Planung von TROIA wurden die Technologien festgelegt, die für die Realisierung von TROIA verwendet werden sollen. Bereits in der Aufgabenstellung wurde bestimmt, dass TROIA mit Hilfe der Programmiersprache Java implementiert wird und auf *Open Source* Produkten basiert.

7.1 Java

Java ist eine objektorientierte, plattformunabhängige Programmiersprache, die von der Firma Sun Microsystems entwickelt wurde. Üblicherweise benötigen Java-Programme zur Ausführung eine spezielle Umgebung (Java Virtual Machine). Der Vorteil ist, dass nur diese Umgebung an verschiedene Computer und Betriebssysteme angepasst werden muss. Sobald dies geschehen ist, laufen auf der Plattform alle Java-Programme ohne Anpassungsarbeiten. Nähere Informationen zu der Programmiersprache Java lassen sich bei [Sun05] finden. Für TROIA wurde die *Java 2 Platform Standard Edition* (J2SE) in der Version 1.4.2 verwendet. Neben der Standard Edition bietet Java auch eine *Java 2 Platform Enterprise Edition* (J2EE) an. Die J2EE enthält zwei interessante APIs zur Erstellung dynamischer HTML Seiten, die Servlets API und die JavaServer Pages (JSP).

7.2 Servlets

Nach [Bodoff02] ist ein Servlet eine Java Klasse, welche die Möglichkeiten von Servern erweitert, auf denen Java Anwendungen laufen, die durch das Request (dt. Anfrage) Response (dt. Antwort) Modell angesprochen werden. Im Prinzip kann ein Servlet auf jeden Anfragetyp antworten. Allerdings werden Servlets häufig für internetbasierte Anwendungen verwendet. Für diese Anwendungen stellt die Java Servlet-Technologie die HTTP-spezifische Servlet-Klasse `javax.servlet.HttpServlet` zur Verfügung.

Servlet-Objekte sind also Instanzen von Java-Klassen, die von der Klasse `javax.servlet.HttpServlet` abgeleitet sind. Ein Webserver kann die Anfragen seines Clients an diese Servlet-Objekte weiterleiten, um sich von ihnen eine Antwort an den Client erzeugen zu lassen. Der Inhalt dieser Antwort wird dabei erst im Moment der Anfrage generiert und ist nicht bereits statisch, etwa in Form einer HTML-Seite, für den Webserver verfügbar.

Die Servlets befinden sich jedoch nicht auf dem Webserver, sondern werden in einem so genannten *Web Container,* auch *Servlet Engine* genannt*,* gehalten. Dieser Webcontainer verwaltet die Servlet-Objekte und stellt sie bei Bedarf bereit. Der Webcontainer seinerseits kommuniziert mit dem Webserver.

Bei Verwendung der Servlet-API und einer entsprechenden Webcontainer-Umgebung besteht die Implementierung einer dynamischen Webseite in Folgendem:

- Es wird eine von `javax.servlet.HttpServlet` abgeleitete Klasse erstellt. Zwei Methoden der Superklasse werden überschrieben, welche für die Verarbeitung der beiden Kernmethoden, `doPost()` und `doGet()`, der HTTP-Spezifikation zuständig sind
- In einer XML Datei, der sogenannten »web.xml«, werden Metainformationen über das Servlet hinterlegt.
- Diese XML-Datei wird zusammen mit der kompilierten Klasse in eine einzige Archiv-Datei, dem *Web-Archiv* (WAR), zusammengeführt.
- Diese WAR-Datei wiederum wird dem Webcontainer übergeben, der die Anwendung erzeugt. Diesen Vorgang nennt man auch *Deployment.*

Servlets generieren HTML-Seiten, indem sie die HTML-Anweisungen mit der Methode `println()` oder Ähnlichem in den Ausgabestrom senden. Dieses Vorgehen hat sich allerdings nicht nur als fehlerträchtig erwiesen, sondern auch die gewünschte Trennung zwischen Geschäftslogik und Visualisierung fällt durch dieses Vorgehen schwer. Denn ändert sich das Erscheinungsbild, so muss das gesamte Programm umgebaut werden. Dabei stecken in vielen dynamischen Programmen oft nur ein oder zwei Zeilen Dynamik. Der Rest ist statischer HTML-Code. Um diesen Problemen entgegenzuwirken, wurde die Technologie der JavaServer Pages entwickelt.

7.3 JavaServer Pages

Die Aufgabe der JavaServer Pages (JSP) ist es, die Erzeugung und Ausgabe von dynamischen HTML-und XML-Daten zu vereinfachen. Wo ein Servlet eine Java-Klasse ist, die sich um die Ausgabe des HTML-Codes kümmert, ist eine JSP eine HTML-Seite die statisches HTML und Java-Code enthält. Dies hat den Vorteil, dass die Logik unabhängig vom Design implementiert werden kann.

Weiterhin bietet die JSP Technologie einen Mechanismus, der es erlaubt dynamische Funktionalitäten in Objekte auszulagern, die über die so genannten *Custom Tags* angesprochen werden können. Diese *Custom Tags* können dann gemeinsam mit den HTML-Tags verwendet werden.

JavaServer Pages werden unter Verwendung eines speziellen JSP-Compilers in Java-Quellcode umgewandelt. Dieser Quellcode, der einem Java-Servlet entspricht, wird im Anschluss durch den Java-Compiler in Bytecode umgewandelt. Die so erzeugten Java-Klassen können dann von einem Webserver mit entsprechender Servlet-Engine ausgeführt bzw. interpretiert werden.

Für die Trennung von Geschäftslogik und Visualisierung empfiehlt [SiStJo02] den Einsatz der Model 2-Architektur bzw. des Model View Controller-Entwurfsmusters (MVC).

7.3.1 Model 2 und Model View Controller

7.3.1.1 Model View Controller

Ziel dieses MVC Entwurfsmusters ist die Trennung bestimmter Programmeigenschaften um flexibles Programmdesign zu erhalten, so dass spätere Änderungen oder Erweiterungen einfach zu handhaben sind und die Wiederverwendbarkeit der einzelnen Komponenten ermöglicht wird. Dafür trennt das MVC-Muster ein Programm in die drei Einheiten Datenmodell (*Model*), Präsentation (*View*) und Programmsteuerung (*Controller*). Abbildung 10 zeigt diese Trennung schematisch.

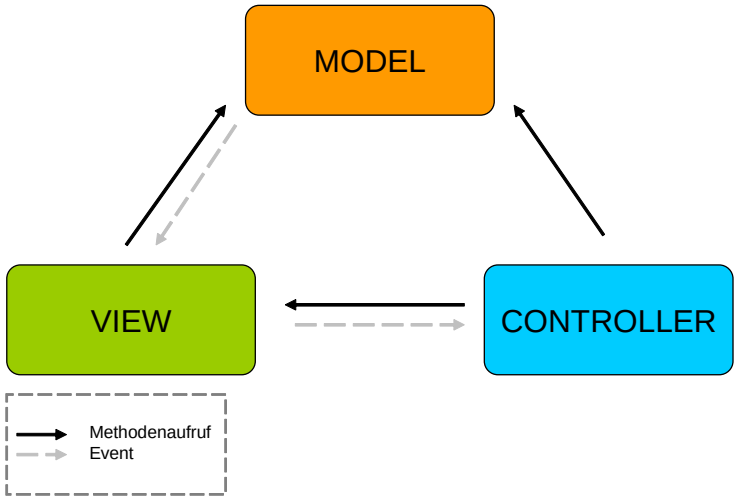

Abbildung 10: Model View Controller Entwurfsmuster

- **Model** – Das Model repräsentiert die Anwendungsdaten und die Businesslogik für den Zugriff und das Ändern der Daten.
- **View** – Die View stellt die Daten dar. Sie greift über das Modell auf die Anwendungsdaten zu und legt fest, wie die Daten dargestellt werden. Die View kennt das Model und ist dort registriert, um sich selbständig aktualisieren zu können.
- **Controller** – Der Controller beinhaltet die Geschäftsintelligenz und bildet mit Hilfe der anderen Schichten die Geschäftsprozesse ab. Er steuert den Ablauf, verarbeitet Daten, entscheidet welche View aufgerufen wird.

7.3.1.2 Model 1, Model 2 und Model 2+

In der Dokumentation sowie in den Diskussionen um Architekturen für die JSP-Verarbeitung trifft man immer wieder auf die Begriffe Model 1 und Model 2. [TuBe03] schreibt, dass diese Begriffe aus der JSP 0.92-Spezifikation stammen. In dieser Version der Spezifikation gab es einen Überblick, der den Abschnitt »JavaServer Pages Access Model« enthielt. Eigentlich gab es nur zwei Modelle, die als Model 1 und Model 2 bezeichnet wurden.

Das Model 1 beschrieb die JSP-Verarbeitung, wie sie damals üblich war. Dabei wurde der HTTP-*Request* direkt an einen JSP gesendet. Die gesamte Request-Verarbeitung erfolgte dann in der JSP oder den mit ihr verbundenen JavaBeans. Nach Beendigung der Verarbeitung erfolgt ein HTTP-Response direkt aus der JSP heraus [TuBe03].

Das Model 2 hingegen setzt das MVC-Entwurfsmuster mit der Hilfe von JSP und Servlets um. Dabei nutzt es die jeweiligen Stärken der beiden Technologien. Die JSPs werden zum Generieren der Präsentationsschicht eingesetzt und die Servlets übernehmen die Steuerung der Prozesse. Nach [Seshadri99] kann man die Model 2-Architektur als serverseitige Implementierung des MVC-Entwurfsmusters sehen.

Jedoch gibt es auch den Ansatz die Präsentationsschicht nicht mit Hilfe von JSP sonder mit XML und XSLT[24] zu generieren. Dieser Ansatz wird oft als Model 2+-Architektur bezeichnet.

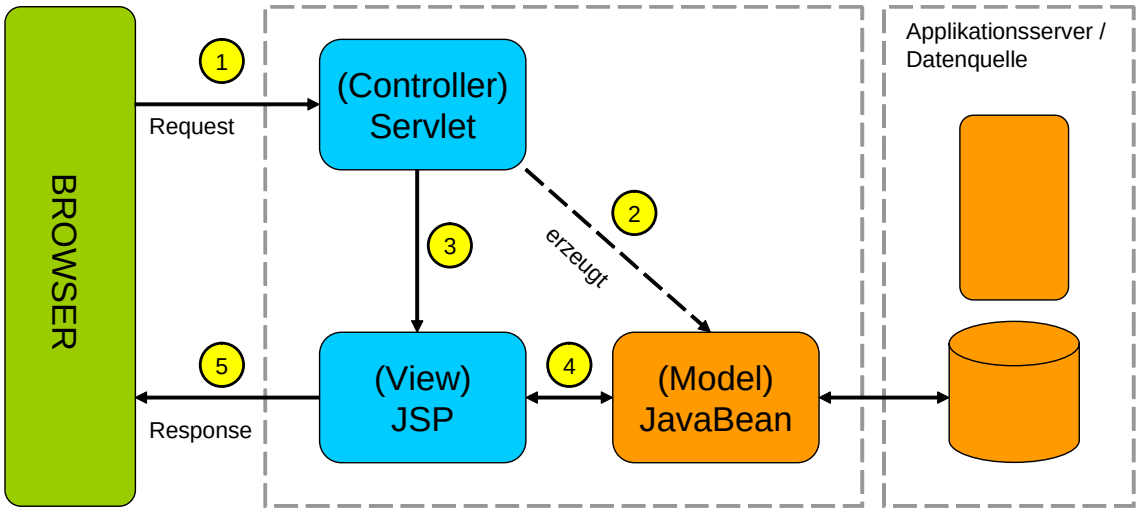

Abbildung 11: Model 2-Architektur

In Abbildung 11 sieht man, dass ein Servlet als Controller fungiert und für die Verarbeitung der Anfragen vom Browser verantwortlich ist. Weiterhin ist dieses Servlet für die Erzeugung der Model-Objekte zuständig, auf die die JSPs zugreifen. Die View wird von JSPs repräsentiert, die je nach Benutzeranfrage vom Controller aufgerufen werden. Die JSPs lesen die benötigten Daten aus dem Model, fügen diese in ihre HTML-*Templates* (dt. Schablonen) ein und senden die so erzeugte, dynamische HTML-Seite an den Browser zurück.

Auf diese Weise erreicht man eine saubere Trennung der Daten von deren Verarbeitung und Darstellung. Im den letzten Jahren sind viele Frameworks entwickelt worden, die diese Architektur umsetzten. Das momentan wohl bekannteste und verbreiteste ist wohl das Jakarta Struts-Framework.

[24] XSLT - Extensible Stylesheet Language Transformation. Siehe auch www.w3.org/TR/xslt.

7.4 Jakarta Struts

Struts ist ein Open Source-Framework für die Präsentationsschicht von Java-Web-Anwendungen. Es gehört zu den Jakarta-Projekten der Apache Software Foundation[25]. Ziel von Struts ist es die Entwicklung von Web-Applikationen zu beschleunigen, indem es die Applikation entsprechend der Model 2-Architektur in einzelne Komponenten gliedert und durch die Implementierung des Front Controller Patterns einen standardisierten Prozess zur Verarbeitung von HTTP-Anfragen bietet.

7.4.1 Struts Controller

Der Controller stellt bei Struts sozusagen das Herz des Frameworks dar. Er steuert den gesamten Arbeitsfluss der Anwendung und stellt die Schnittstelle dar, über die man das Struts-Framework in die eigene Anwendung einbinden kann. Wie bereits erwähnt, wird der Controller durch die Implementierung des Front Controller Patterns realisiert. Der Front Controller ist in Struts ein Servlet, das grundsätzlich alle Anfragen entgegen nimmt und aufgrund seiner Konfiguration entscheidet, an welche Komponente die Anfrage weitergeleitet wird. Die Konfiguration des Front Controllers ist in der »struts-config.xml« hinterlegt.

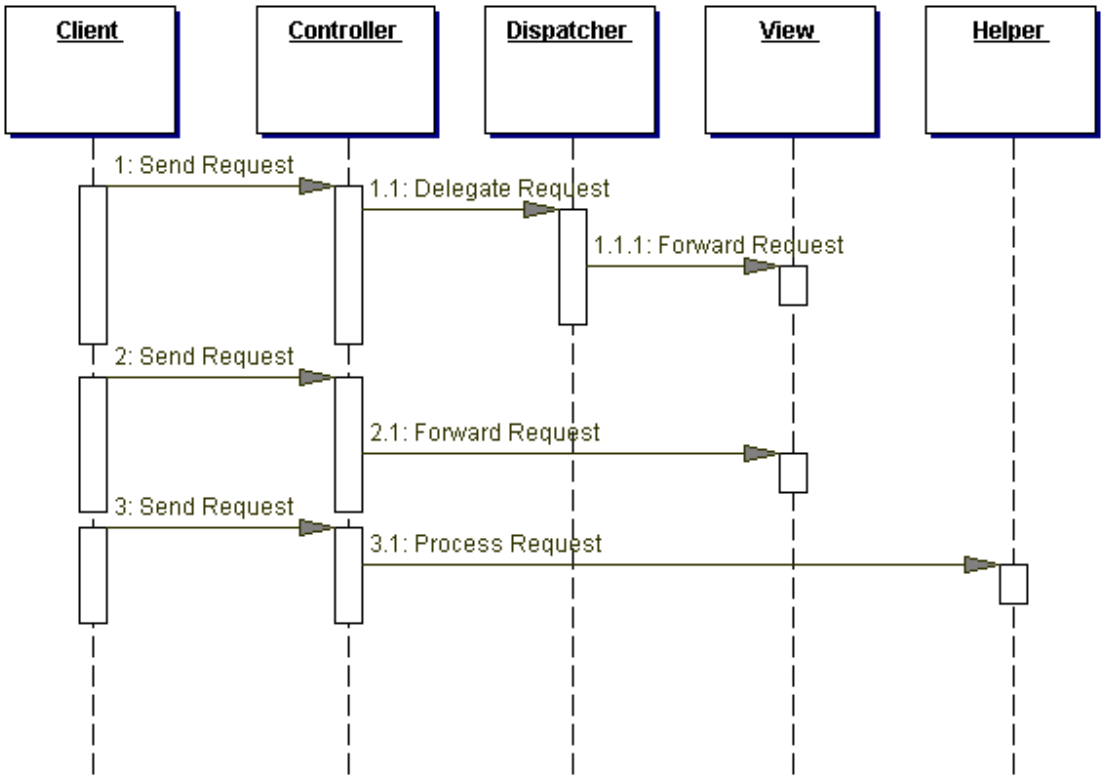

Abbildung 12: Sequenzdiagramm Front Controller

[25] Für mehr Informationen über die Apache Software Foundation siehe www.apache.org.

Das Sequenzdiagramm aus Abbildung 12 zeigt die Funktionsweise des Front Controller Patterns. Der Front Controller nimmt die Anfrage von den Clients entgegen und leitet diese entweder an einen Dispatcher weiter oder er bearbeitet die Anfrage selbst. Der Dispatcher entscheidet anhand der Konfiguration aus der »strut-config.xml« über die weitere Verarbeitung und gibt die Anfrage an die entsprechenden Modelkomponenten weiter. Anschließend ruft er die View auf.

7.4.2 Struts View

In der Regel werden für die Realisierung der View in Struts JavaServer Pages eingesetzt. Zwar kann die View auch durch die Model 2+-Architektur erzeugt werden, allerdings bietet das Struts Framework nur wenig Unterstützung für das Erzeugen von XML-basierten Views.

Die bevorzugte Verwendung von JSPs in Struts Anwendungen ist sicher auch dadurch begründet, dass Struts umfangreiche *Tag*-Bibliotheken zur Verfügung stellt. Diese *Tag*-Bibliotheken enthalten die bereits erwähnten *Custom Tags*. Diese *Tags* kapseln Funktionalität und verlagern diese in eigene Klassen. Dadurch wird Java Code in den JSPs vermieden.

Für Internationalisierung von Oberflächen bietet Struts das *Tag* `<bean:message key="schlüsselname"/>`. Dieses Tag hat eine sehr ähnliche Funktionsweise wie das bereits beschriebene OSE Tag `<ose:text>`. Auch hier wird über das `PropertiesResourceBundle` auf die lokalisierten Properties zugegriffen, so dass Texte entsprechend dem *Locale* des Benutzers angezeigt werden können.

7.4.3 Struts Modell

Das Model wird in Struts in der Regel durch JavaBeans realisiert. Es können aber auch andere Java-Klassen verwendet werden. In den Model-Klassen wird die Geschäftslogik implementiert. Außerdem regeln sie den Zugriff auf entfernte Systeme, wie z.B. auf eine Datenbank oder einen Applikationsserver.

7.5 Apache Ant

Auch Ant ist ein Open Source Projekt der Apache Software Foundation. Das Akronym Ant steht für "Another Neat Tool" (dt. Noch ein hübsches Werkzeug). Das Programm Ant ist ein in Java geschriebenes Werkzeug zum automatisierten Erzeugen von Programmen aus Quell-Code. Dieses Werkzeug wird sowohl zum Bauen der IUCCA-Instanzen als auch von TROIA verwendet.

Gesteuert wird Ant durch eine XML-Datei, dem so genannten *Build-Script*. In dieser wird zunächst ein *Projekt* definiert, welches *Targets* (dt. Ziele) enthält. Diese sind vergleichbar mit Funktionen in Programmiersprachen und enthalten unter anderem Aufrufe von *Tasks* (dt. Aufgaben). Ein *Task* ein ist elementarer Arbeitsschritt. Zwischen den

Targets können und sollten Abhängigkeiten definiert werden. Nach dem Starten arbeitet Ant die einzelnen *Targets* entsprechen ihrer Abhängigkeiten ab.

```
<project name="IUCCA" default="build" basedir=".">

  <target name="build" depends="ini,compile,jar ">
    <echo message="IUCCA wurde erfolgreich gebaut" />
  </target>

  <target name="init" >
    <mkdir dir="${bin}"/>
    <mkdir dir="${jardir}"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${src}" destdir="${bin}" ></javac>
  </target>

  <target name=„jar" depends="init,compile">
    <jar destfile="${jarfile}" basedir="${bin}"></jar>
  </target>

</project>
```

Abbildung 13: Beispiel eines Ant Build-Script

Abbildung 13 zeigt ein sehr vereinfachtes Ant *Build-Script*. Man erkennt, dass es in dem Projekt vier von einander abhängige *Targets* gibt. So wird das *Target* build erst ausgeführt nachdem alle *Targets,* von denen es abhängig ist, erfolgreich ausgeführt wurden. Wenn ein Target aufgerufen wird, werden die *Tasks,* die es enthält abgearbeitet. Das Target compile ruft zum Beispiel den *Task* javac auf, der für das Kompilieren des Quellcodes zuständig ist.

In IUCCA wird durch diese *Build-Scripts* festgelegt, welche Ressourcen aus den Länder- und Sprachpaketen gemeinsam mit dem Systemkern zu einer länderspezifischen Instanz zusammengeführt werden. Auch der TROIA-Parser wird in Zukunft aus den *Build-Scripts* heraus aufgerufen.

7.6 Apache Tomcat

Da sowohl Servlets als auch JSPs einen Web-Container benötigen, wird für den Betrieb von TROIA der Apache Tomcat Servlet-Container verwendet. Auch Apache Tomcat ist ein Open Source Project der Apache Software Foundation. Weiterhin stellt er die Referenzimplementierung der Servlet- und JSP-Spezifikationen dar.

Der Tomcat stellt also eine Umgebung (Container) bereit, in der Java Servlets betrieben werden können. Zu Tomcat gehört auch der Jasper JSP-Compiler, der JSPs in Servlets übersetzt

Zusätzlich bietet Tomcat noch weitere Funktionen, wie etwa Authentifizierungs- und Authorisierungsmöglichkeiten, bietet einen internen Verzeichnisdienst über JNDI[26] an und eröffnet die Möglichkeit gepoolte Datenquellen im Container zu hinterlegen, die dann von den darin laufenden Applikationen verwendet werden können.

Tomcat hat zwar einen eigenen HTTP-Server, aber der sollte nur für Entwicklungszwecke verwendet werden. In TROIA wird Tomcat zusammen mit dem Apache Web-Server[27] verwendet. Dabei leitet der Web-Server lediglich alle Anfragen an TROIA an den Web-Container weiter.

7.7 MySQL

Als Datenquelle wird in TROIA die MySQL-Datenbank verwendet. Dies ist ein Open Source Produkt der MySQL AB[28]. Streng genommen ist MySql keine relationale Datenbank, da sie in dem Standardtabellenformat MyISAM weder referenzielle Integrität noch Transaktionen unterstützt. Dazu [WikiMySQL05]:

> *»Allerdings ist sie [MySQL Datenbank] im Gegensatz zu Oracle, DB2 oder PostgreSQL keine relationale Datenbank. Die Daten werden zwar in zweidimensionalen Tabellen gespeichert und können mit einem Primärschlüssel versehen werden. Es ist aber keine Definition eines Fremdschlüssels möglich. Der Benutzer ist somit bei einer MySQL-Datenbank völlig allein für die referenzielle Integrität der Daten verantwortlich. Allein durch die Nutzung externer Tabellenformate wie InnoDB oder Berkeley DB wird eine Relationalität ermöglicht.«*

Seit der MySQL Version 4.0 ist das InnoDB Tabellenformat jedoch in MySQL enthalten. Als Standard wird allerdings auch weiterhin das MyISAM Format benutzt.

Ab der Version 4.1 erlaubt MySql auch das Speichern von UTF-8 Daten. Dafür kann man den Zeichensatz entweder für den gesamten Server, eine Datendank, eine Tabelle oder für eine einzelne Spalte festlegen. Für TROIA wurde für die Tabelle `texts`, in der die Schlüssel-/Wert-Paare aus den Properties-Dateien gespeichert werden, der Zeichensatz UTF-8 festgelegt.

---

[26] JNDI - Java Naming and Directory Interface. Mithilfe von JNDI können Daten und Objektreferenzen anhand eines Namens abgelegt und später über den Namen wieder gefunden werden

[27] Siehe http://httpd.apache.org

[28] Siehe http://www.mysql.com/

8 Realisierung von TROIA

Nachdem die Anforderungen an TROIA analysiert und der Aufbau des Gesamtsystems sowie die verwendete Technologie festgelegt wurden, konnte mit dem Entwurf der Architektur für TROIA begonnen werden.

TROIA besteht aus zwei Komponenten, der webbasierten Anwendung TROIA und dem TROIA-Parser. Als erstes wird nun der TROIA-Parser beschrieben. Anschließend wird die Webanwendung TROIA erläutert.

8.1 TROIA-Parser

Eine der Anforderungen war, dass TROIA einem Übersetzer die Möglichkeit bietet, einen zu übersetzenden Begriff in dem Kontext zu sehen, in welchem er in der IUCCA-Oberfläche verwendet wird. Um diese Anforderung zu realisieren, werden zusätzlich zu den Schlüssel-Wert-Paaren aus den Properties-Dateien noch Informationen über die Verwendung der Schlüssel in den JSP-Dateien benötigt. Die Aufgabe des TROIA-Parsers ist es, diese Daten zu generieren und gemeinsam mit den Daten der Properties-Dateien in eine Datenbank zu schreiben.

8.1.1 Der TROIA-Parser als Ant Task

Wie in dem *Deployment*-Diagramm in Abbildung 9 zu sehen ist, befindet sich der TROIA-Parser als JAR-Datei auf dem Test- bzw. Integrationsserver. Dort kann er durch ein Ant Build-Script gestartet werden.

Ant bietet Anwendern die Möglichkeit, eigene *Tasks* zu implementieren und in das Build-Script einzufügen. Dafür muss die Ant-Klasse `org.apache.tools.ant.Task` erweitert und die abstrakte Methode `execute` implementiert werden. Im TROIA-Parser geschieht dies durch die Klasse `de.tsystems.ec.troia.parser.ParserTask`. Nachdem die Ant-Klasse erweitert wurde, kann man im Ant *Build-Script* einen neuen *Task* definieren und aufrufen.

```
<project name="IUCCA" default="build" basedir=".">
  <target name="build" depends="ini,compile,jar, parse ">
    <echo message="IUCCA wurde erfolgreich gebaut" />
  </target>
  <target name="init" >
    <mkdir dir="${bin}"/>
    <mkdir dir="${jardir}"/>
  </target>
  <target name="compile" depends="init">
    <javac srcdir="${src}" destdir="${bin}" ></javac>
  </target>
  <target name=„jar" depends="init,compile">
    <jar destfile="${jarfile}" basedir="${bin}"></jar>
  </target>
  <target name="initParser" depends="Jar">
     <taskdef name="parser"
       classname="de.tsystems.ec.troia.parser.ParserTask"
       classpath="${taskjar}" />
  </target>
  <target name="parse" depends="initParser" >
     <parser targetversion="4.6" country="ES"
       basedir="/opt/iucca/3001" />
  </target>
</project>
```

Abbildung 14: Definition und Aufruf eines Ant-*Tasks*

Abbildung 14 zeigt die Definition und Verwendung eines benutzerdefinierten Ant-*Tasks*. In dem *Target* `initParser` wird der neue *Task* definiert. Dabei wird dem *Task* ein Name zugewiesen. Weiterhin wird der Klassenpfad und der Klassenname der Klasse bekannt gegeben, die die Ant-Klasse `org.apache.tools.ant.Task` erweitert. Nun kann der neue *Task* in einem *Target* verwendet werden. Dabei ist es möglich, dem *Task* Parameter zur Verarbeitung zu übergeben. Im Fall des TROIA-Parsers sind dies drei Parameter, die Zielversion, das Land und der Pfad, in dem die neue IUCCA-Instanz gebaut wurde.

8.1.2 Funktionsweise des TROIA-Parsers

Nachdem der Parser aus dem *Build-Script* heraus gestartet wurde und die Informationen über Zielversion, Land und Pfad erhalten hat, durchsucht er das angegebene Verzeichnis nach JavaServer Pages. Sobald er eine JSP gefunden hat, wird diese auf das OSE-Tag `<ose:text>` hin analysiert. Anschließend wird geprüft, ob von der aktuellen JSP andere JavaServer Pages eingebunden werden. Sollte dies der Fall sein werden auch diese Seiten analysiert, wobei die gefundenen Schlüssel zu der aktuellen Seite gezählt werden.

Für die Verwendung des OSE-*Tags* `<ose:text>` gibt es im IUCCA-Team die Konvention, dass sowohl der Schlüssel als auch dessen Wert angegeben wird. Dadurch wird sichergestellt, dass ein Text auch dann angezeigt werden kann, wenn ein Schlüssel nicht in den Properties gefunden wurde. Das *Tag* sieht folgendermaßen aus:

<ose:text key="salesman">Verkäufer</ose:text>.

Sobald ein solches *Tag* gefunden wurde, schreibt der Parser den Seitennamen, den Schlüssel und dessen Wert in ein JavaBean. Jedoch muss vor dem Speichern des Wertes geprüft werden, ob dieser HTML-Escape-Sequenzen enthält. Diese werden vor dem Speichern in die entsprechenden Unicode-Zeichen umgewandelt. Weiterhin ermittelt der Parser, auf wie vielen Seiten ein bestimmter Schlüssel verwendet wird. Dies hätte auch über eine Datenbankabfrage in der Webanwendung TROIA geschehen können. Allerdings hätte dieses Vorgehen zu einer großen Zahl von Datenbankabfragen beim Erzeugen einer Seite geführt.

Sobald alle JSPs analysiert sind, werden die Properties-Dateien gesucht und eingelesen. Zu jeder Properties-Datei wird auch deren Name und *Locale* gespeichert. Anschließend vergleicht der Parser die Default-Properties mit den Daten, die aus den JSPs ermittelt wurden. Kann der Parser ein Schlüssel-/Wert-Paar, welches er den JSPs ermittelt hat, nicht in den Default-Properties finden, fügt er dieses Paar den Default-Properties hinzu.

Schlüssel, die sich in den Properties-Dateien befinden, aber nicht in den JSPs gefunden wurden, können hingegen nicht gelöscht werden. Grund dafür ist die Möglichkeit, dass sie von der Anwendung dynamisch eingebunden werden. Das bedeutet, dass Entwickler neue Schlüssel, die sie in den JSPs verwenden, nicht mehr in die Properties-Dateien einpflegen müssen. Schlüssel, die nicht mehr benötigt werden, müssen aber manuell aus den Default-Properties gelöscht werden.

Nachdem die Default-Properties aktualisiert wurden, vergleicht der Parser diese mit den lokalisierten Properties. Dabei werden die in den Default-Properties neu hinzu gekommenen Schlüssel eingefügt und die weggefallenen Schlüssel gelöscht. Somit müssen die Entwickler nur noch die Default-Properties pflegen und die lokalisierten Properties werden generiert.

Wie in Kapitel 4.4.2.5 bereits beschrieben, sind die Properties-Dateien in ISO Latin 1 kodiert und alle Zeichen außerhalb dieses Zeichensatzes liegen in Unicode-Escape-Zeichen vor. Da der Parser für das Lesen der Properties-Dateien nicht das `PropertiesResourceBundle` sondern einen I/O-Zugriff benutzt, muss er die Escape-Zeichen in Unicode umwandeln, bevor er die Werte in die Datenbank schreibt. Das `PropertiesResourceBundle` kann nur verwendet werden, wenn sich die Properties im Klassenpfad der Anwendung befinden Dann werden jedoch die Escape-Zeichen vom `PropertiesResourceBundle` automatisch umgewandelt.

Im letzten Bearbeitungsschritt, prüft der TROIA-Parser, ob der Wert eines Schlüssels Verweise auf andere Schlüssel enthält. Ist dies der Fall, so löst der Parser die Verwei-

se auf und speichert den zusammengesetzten Text. Durch dieses Feld wird die Performance der Volltextsuche in TROIA gesteigert. Anschließend werden die Daten in die Datenbank geschrieben und können über TROIA bearbeitet werden.

8.2 Die Webanwendung TROIA

8.2.1 Die Architektur von TROIA

Die webbasierte Anwendung ist die zweite Komponente von TROIA. Über das Web-Interface von TROIA können Benutzer die Daten bearbeiten, die vom TROIA-Parser generiert und in die Datenbank geschrieben wurden. TROIA wurde nach der für Webanwendungen typischen Mehrschichtarchitektur entworfen. Ziel dieser Architektur ist die Aufteilung einer Anwendung in Module, die auf physisch getrennten Knoten betrieben werden können. Dadurch erreicht man eine bessere Wartbarkeit und Skalierbarkeit der Anwendung. Außerdem können so die Module wieder verwendet werden.

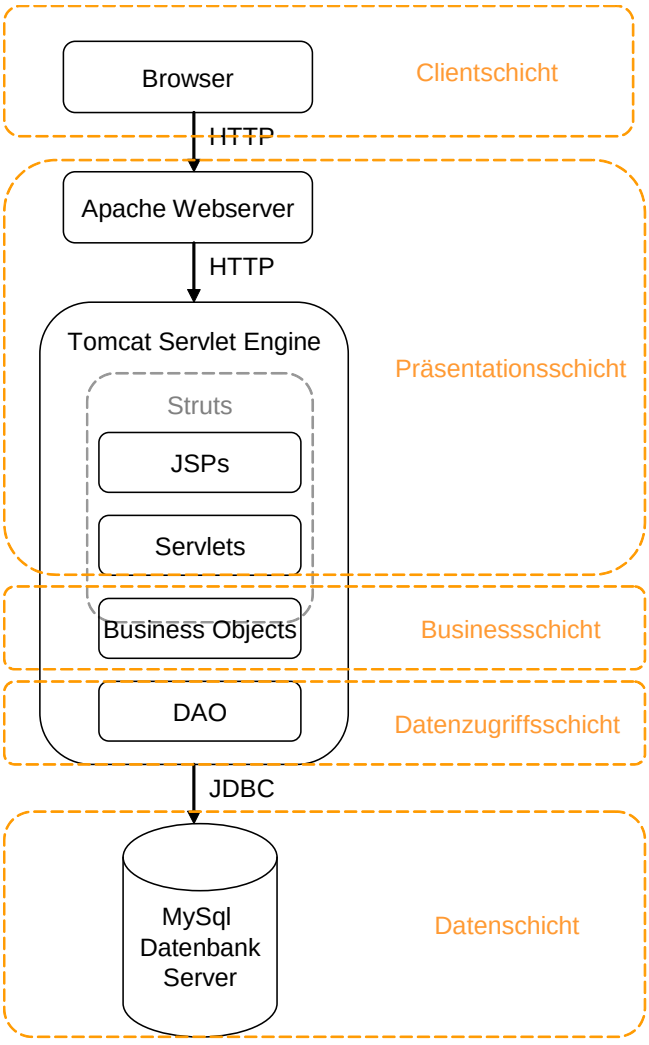

Abbildung 15: TROIA Systemarchitektur

Wie man in Abbildung 15 sieht, kann man TROIA in fünf Schichten einteilen.

- Die Client-Schicht repräsentiert den Anwender. Dieser kann über einen Browser auf die Anwendung zugreifen.
- Die Präsentationsschicht enthält zum Einen den Webserver, der lediglich Anfragen vom Browser an den Servlet-Container weiterleitet. Im Servlet-Container werden die Anfragen vom Struts Controller entgegen genommen und zur Verarbeitung an die Business-Schicht delegiert. Anschließend wird über eine JSP die Antwort an den Browser geschickt. Diese Schicht beinhaltet also den Controller und die View des MVC-Patterns.
- Die Business-Schicht enthält die eigentliche Anwendungslogik von TROIA. Sie empfängt Anfragen aus der Präsentationsschicht, verarbeitet diese und greift über die Datenzugriffsschicht auf die Datenbank zu. Die Business-Schicht stellt zusammen mit der Datenzugriffsschicht und der Datenschicht das Modell des MVC-Entwurfsmusters dar.
- Die Datenzugriffsschicht kapselt den Zugriff auf die Datenbank, dadurch ist es möglich die Logik für den Zugriff zu verändern ohne das die restlichen Schichten davon betroffen werden.
- Datenschicht enthält die Datenbank, in der sich die Daten der Sprachressourcen befinden.

In der nachfolgenden Abbildung 16 werden die einzelnen Pakete aus denen TROIA besteht gezeigt.

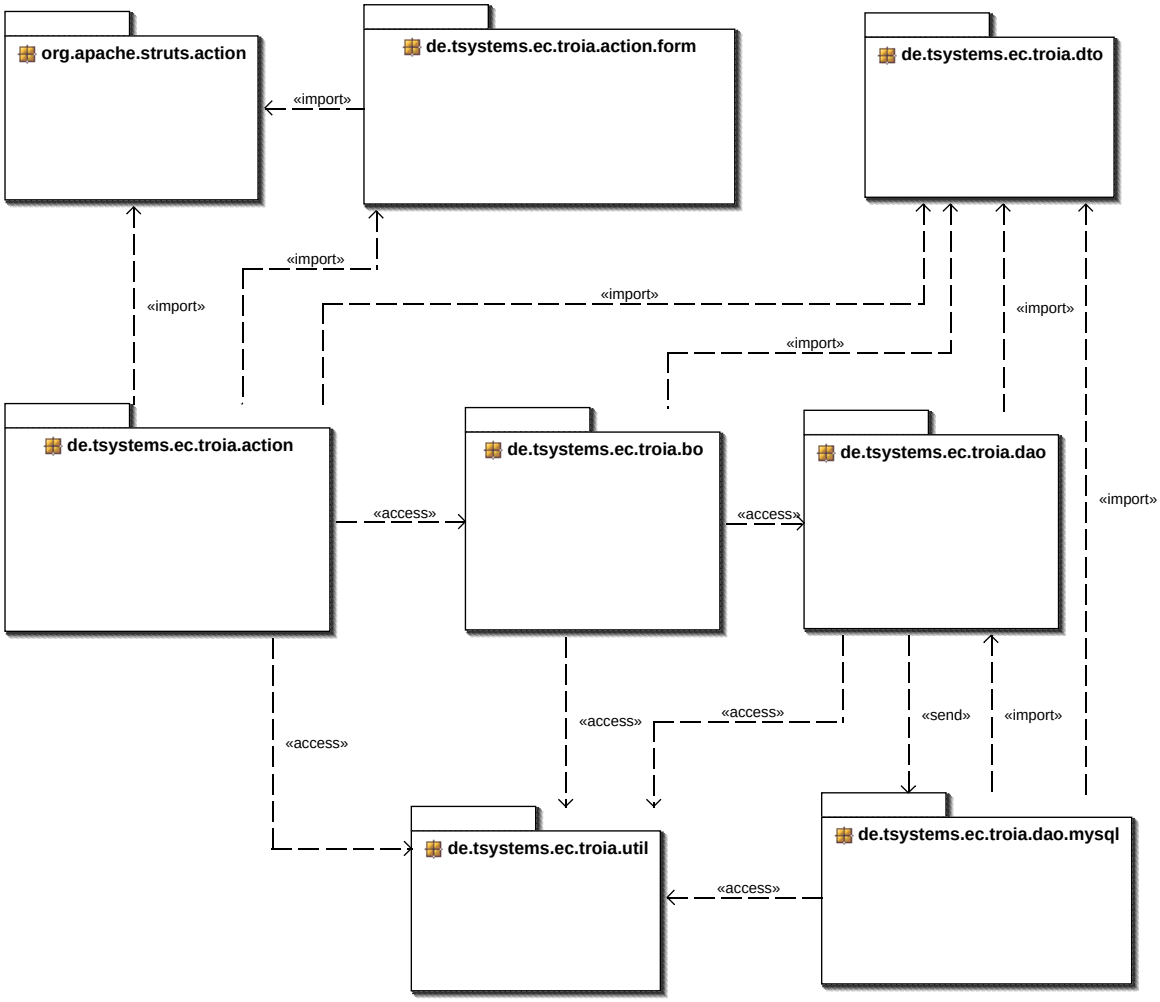

Abbildung 16: Übersicht TROIA Pakete

Den Einstiegspunkt in die Anwendung bildet das `org.apache.struts.action`-Paket. In diesem Paket befinden sich die Klassen des Struts-Frameworks, die den Controller darstellen. Durch das Erweitern der Struts-Klasse `Action` wird das Framework in TROIA eingebunden. In dem Paket `de.tsystems.ec.troia.action` befinden sich die TROIA-Klassen, welche die Struts-Klassen erweitern. An diese Klassen leitet Struts die Benutzeranfragen weiter und sie entscheiden, wie die Anfragen verarbeitet werden und welche *Business Objects* aufzurufen sind. Die *Business Objects* werden in dem Pakte `de.tsystems.ec..troia.bo` zusammengefasst und implementieren die Anwendungslogik von TROIA.

Das Paket `de.tsystems.ec.troia.action.form` beinhaltet Klassen, die von der Struts Klasse `ActionForm` abgeleitet sind. Mit Hilfe dieser Klassen überträgt Struts die Daten aus HTML-Formularen an die `Action` Klassen.

Der Zugriff auf die Datenbank wird durch die Pakete `de.tsystems.ec.troia.dao` und `de.tsystems.ec.troia.dao.jdbc` gekapselt. Die Daten werden mit Hilfe von *Transfer Objects,* die im Paket `de.tsystems.ec.troia.dto` gesammelt wurden, innerhalb der Anwendung weiter gegeben,. In `de.tsystems.ec.troia.util` werden Hilfsklassen z.B. für das Umwandeln von Unicode in Java Escape-Sequenzen bereitgestellt. Im folgenden Abschnitt werden die Pakete den einzelnen Schichten zugeordnet und erklärt.

8.2.2 Die Schichten im Detail

8.2.2.1 Die Client-Schicht

Die Client-Schicht wird durch die den Browser repräsentiert, mit welchem der Anwender auf die Präsentationsschicht von TROIA zugreift. Bei der Entwicklung der HTML-Oberflächen wurde darauf geachtet, dass diese von allen gängigen Browsern fehlerfrei dargestellt werden können.

8.2.2.2 Die Präsentationsschicht

Diese Schicht wird durch den Web-Server und den Servlet-Container realisiert. In TROIA werden keine Komponenten der Anwendung auf dem Web-Server abgelegt. Der Web-Server leitet lediglich die Anfragen aus der Client-Schicht an den Servlet-Container weiter. Die Anfragen werden vom Struts-Controller entgegen genommen und weiter delegiert. Der Controller besteht aus den Klassen `ActionServlet`, `RequestProcessor` und `Action` des `org.apache.struts.action`-Pakets. Das UML-Klassendiagramm in Abbildung 17 zeigt die in dem Paket enthaltenen Klassen. Die Klasse `ActionServlet` ist der Front Controller, der die Anfragen des Client annimmt und an den `RequestProcessor` weiter leitet. Der `RequestProcessor` analysiert die Anfrage und gibt sie an die entsprechende `Action` Klasse weiter.

Um das Struts Framework in die eigene Applikation einzubinden, wird die Klasse `Action` erweitert und die Methode `execute()` überschrieben. In dem Paket `de.tsystems.ec.troia.action` wurden alle TROIA-Klassen zusammengefasst, welche die Struts Action-Klasse erweitern. Durch diese von `Action` abgeleiteten Klassen werden alle in TROIA möglichen Benutzeranfragen abgebildet. Sie verarbeiten diese Anfragen und greifen auf die benötigten Methoden aus den entsprechenden *Business Objekten* zu. Anschließend rufen sie die gewünschte JSP auf und stellen dieser alle benötigten Daten zur Verfügung.

Da es in einer Anwendung üblicherweise mehr als eine benutzerdefinierte `Action` gibt, braucht der `RequestProcessor` Informationen darüber, welchen *Request* er an welche `Action` weiterleiten soll. Diese Informationen werden in Struts in einer zentralen XML-Konfigurationsdatei abgelegt. Diese Datei trägt den Namen »struts-config.xml«. In dieser Datei wird festgelegt, wie sich die Komponenten der Applikation zusammenfügen und wie diese Komponenten verwendet werden sollen.

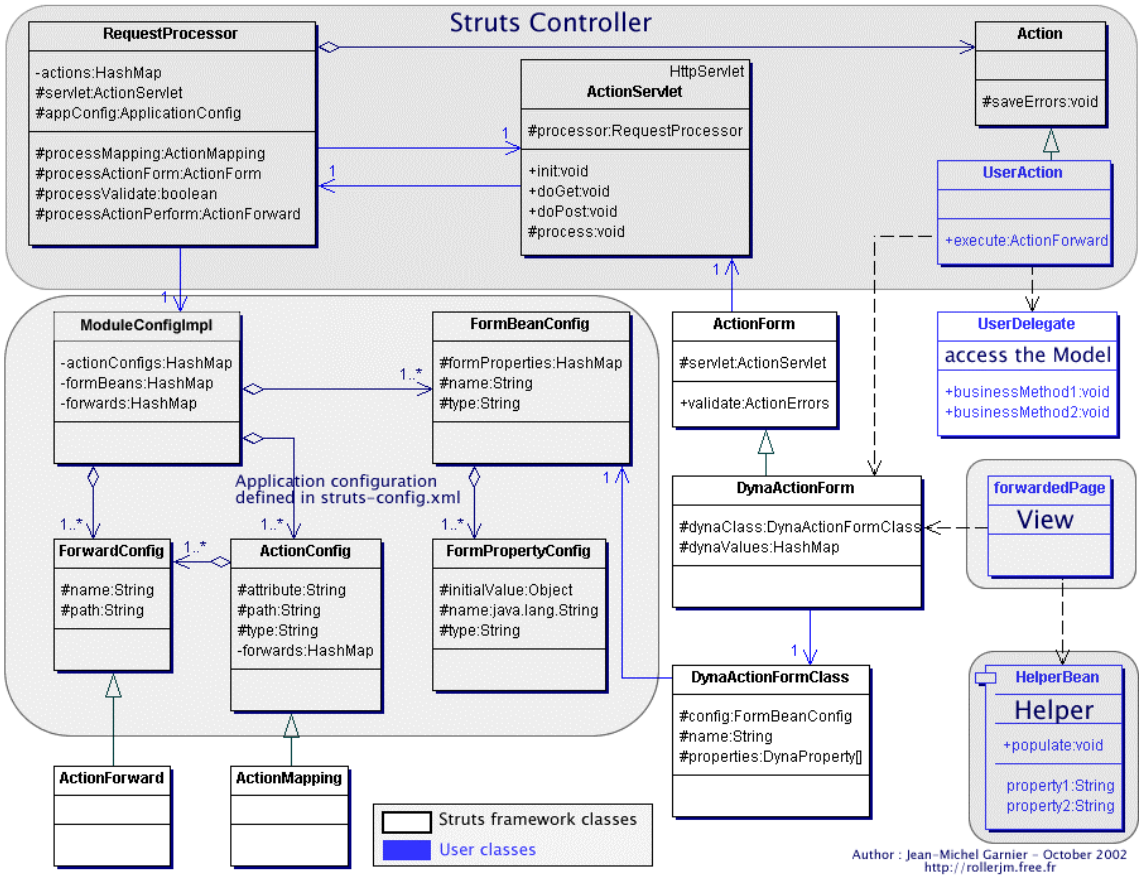

Abbildung 17: Struts Controller UML-Diagramm. [Garnier03].

Aus Abbildung 17 ist ersichtlich, dass der `RequestProcessor` auf die Klasse `ModulConfigImpl` zugreift und dass diese Klasse die Klassen `ForwardConfig`, `ActionConfig`, `FormBeanConfig` und `FormPropertyConfig` aggregiert. Diese Klassen repräsentieren die Daten aus der Konfigurationsdatei.

In Abbildung 18 wird ein kleiner Ausschnitt aus der »struts-config.xml« Konfigurationsdatei für TROIA gezeigt.

Unterhalb des XML-Tag `<action-mapping>` werden in `<action>` die einzelnen benutzerdefinierten `Action`-Klassen angegeben. Diese Daten werden dann durch die Klasse `ActionConfig` repräsentiert.

- Das Attribut `path` definiert den Namen der `Action`, der bei einer Anfrage benutzt wird.
- Das Attribut `type` beschreibt den Klassennamen der benutzerdefinierten `Action`-Klassen. Bei einer Anfrage in der Form http://<url>/troia/loadText.do kann der Controller über den `path` »loadText.do« die entsprechende Klasse vom `type LoadTypeAction` aufrufen.

- Das Attribut `name` gibt den Namen des Form-Beans an, das dieser `Action` zugeordnet ist und mit dessen Hilfe Daten aus den HTML-Formularen an die `Action`-Klassen weiter gegeben werden.
- Durch das Tag `<forward>` wird der `Action` der Name und der Pfad der JSP angeben, an die sie den Kontrollfluss weiterleiten soll.

```
<struts-config>
<!-- ==== Form Bean Definitions ================================== -->
  <form-beans>
    <form-bean
      name="loadTextDynaForm"
      type="org.apache.struts.action.DynaActionForm">
      <form-property name="field" type="java.lang.String"/>
      <form-property name="textList" type="java.util.ArrayList" />
    </form-bean>
  </form-beans>
<!-- ==== Action Mapping Definitions ============================= -->
  <action-mappings>
    <action
      path="/loadText"
      type="de.tsystems.ec.troia.action.LoadTextAction"
      name="loadTextDynaForm"
      <forward name="success" path="/pages/text.jsp" />
    </action>
  </action-mappings>
<!-- ==== Message Resource Definition ============================== -->
  <message-resources parameter="de.tsystems.ec.troia.resources.message"/>
</struts-config>
```

Abbildung 18: Struts Konfigurationsdatei »struts-config.xml«

In dem `<form-bean>` *Tag* werden die benutzerdefinierten `ActionForm`-Klassen beschrieben. Dabei muss man zwischen den normalen `ActionForm` und den generischen `DynaActionForm` unterscheiden. Normale `ActionForm`-Klassen werden durch die Attribute `name` und `type` beschrieben.

- Das Attribut `name` definiert den Namen des Form-Bean unter welchem es von einer `Action` aber auch von JSPs angesprochen werden kann.
- Das Attribut `type` beschreibt den Klassennamen der benutzerdefinierten `ActionForm`-Klasse.

Eine `ActionForm` ist also eine Klasse, die vom Benutzer erweitert und implementiert wird. Eine `DynaActionForm`-Klasse hingegen wird nur in der Konfigurationsdatei definiert und dann vom Framework generiert. Dadurch lassen sich bequem neue Klassen hinzufügen oder bestehende modifizieren, ohne das der Programmcode geändert werden muss.

- Das Attribut `type` muss bei dynamischen `ActionForm`-Klassen immer auf die Klasse `org.apache.struts.action.DynaActionForm` verweisen.
- Mit dem Tag `<form-propertie>` werden der `DynaActionForm`-Klasse neue Attribute zugewiesen.

Ein weiterer Bestandteil der Präsentationsschicht sind die JSPs, die den *Actions* durch das Tag `<forward>` bekannt gemacht werden. Nachdem die *Actions* die Verarbeitung der Anfrage durch die Business-Schicht veranlasst haben, geben sie die Anfrage an die einsprechende JSP weiter und erzeugen so die View.

Die JSPs von TROIA wurden mit Hilfe des Struts-*Tag* `<bean:message>` internationalisiert. Weiterhin ist darauf geachtet worden, dass kein Javacode in den JSPs verwendet wurde. Alle dynamischen Elemente, z.B. das Erzeugen von Listen, wurden durch die *Custom Tags* der Struts *Tag*-Bibliotheken realisiert.

8.2.2.3 Die Business-Schicht

Die Business-Logik von TROIA, das heißt die Logik für die Bearbeitung der Sprachressourcen, wurde in *lightweight* (dt. leichtgewichtig) *Business Objects* (BO) gekapselt. In einer Anwendung, die einen Applikationsserver verwendet, werden die *Business Objects* üblicherweise in *Entity Beans*[29] implementiert, um die Sicherheits- und die Transaktions-Management-Funktionen des Applikationsservers zu nutzen. Da für TROIA allerdings nur ein Servlet-Container eingesetzt wird, sind die *Business Objects* als sogenannte »Pure Old Java Objects« (POJO) [Gupta05], also als einfache Java Objekte, implementiert. Die für TROIA verwendeten *Business Objects* sind zustandslos und verwenden JDBC[30]-Technologien zur Persistenzverwaltung.

Business Objects repräsentieren innerhalb einer Anwendung »Dinge« der realen Welt. In TROIA sind diese zum einen die Sprachressourcen, die durch die Klasse `TextBO` vertreten werden. Weiterhin existieren Klassen für die IUCCA-Seiten und die TROIA-Konfiguration. Schließlich gibt es noch eine Klasse, welche die Benutzer repräsentiert. Diese Klassen stellen den *Actions* aus der Präsentationsschicht die benötigten Methoden zum Laden, Bearbeiten und Speichern von Daten zur Verfügung. Für das Laden und Speichern von Daten benutzen die *Business Objects* die Datenzugriffsschicht. Der Datenaustausch zwischen den Schichten erfolgt durch die *Transfer Objects* aus dem

[29] Entity Beans sind Teil des J2EE Standards und repräsentieren die persistenten Daten eines Systems.

[30] JDBC – Java Database Connectivity ist eine API der Java-Plattform die eine einheitliche Schnittstelle zu Datenbanken verschiedener Hersteller bietet.

Paket `de.tsystems.ec.troia.dto`. Diese Klassen entsprechen dem *Transfer Object* Entwurfsmuster [AlCrMa01]. In TROIA sind die *Transfer Objects* einfache JavaBeans, die ihre Daten über `get()` und `set()` Methoden anderen Klassen zur Verfügung stellen.

Durch die Kapselung der Anwendungslogik innerhalb der Business-Schicht wird die Flexibilität und Wiederverwendbarkeit einer Anwendung gesteigert. So ist es z.B. möglich die aktuelle Präsentationsschicht gegen eine andere, z.B. eine Swing-GUI auszutauschen, ohne Änderungen in der Businesslogik vornehmen zu müssen.

8.2.2.4 Die Datenzugriffsschicht

Die Aufgabe dieser Schicht ist es, den Datenzugriff vor der restlichen Anwendung zu verbergen. Dadurch lässt sich die Implementierung des Datenzugriffs an einer zentralen Stelle ändern, statt die Veränderungen in der gesamten Software vornehmen zu müssen. So kann man problemlos das Datenmodell ändern, eine andere Datenbank verwenden oder ein Persistenzframework einbinden.

In TROIA wurde die Datenzugriffsschicht durch die Verwendung des *Data Access Objects* (DAO)-Entwurfsmusters [AlCrMa01] realisiert. Die DAOs bieten alle Methoden, die für das Lesen, Schreiben und Ändern von Daten benötigt werden. In TROIA wurde für jede Datenbanktabelle ein DAO implementiert.

Die Abbildung 19 zeigt das UML-Klassendiagramm der Datenzugriffsschicht. Die Klasse `DAOFactroy` ist eine dem gleichnamigen Entwurfsmuster entsprechende *Abstract Factory* [GoF04]. Sobald ein *Business Object* auf die Datenbank zugreifen möchte, holt es sich über die abstrakte `DAOFactory`-Klasse die Instanz einer konkreten *Factory*-Klasse. Die `DAOFactory` entscheidet welche DAO-Implementierung benötigt wird und erzeugt eine konkrete *Factory*–Klasse, die dem *Factory Method*-Entwurfsmuster [GoF04] entspricht.

Momentan wird in TROIA über JDBC auf eine MySQL-Datenbank zugegriffen. Daher liefert die `DAOFactory` eine Instanz der `MySqlFactory`-Klasse. Diese konkrete *Factory*-Klasse erzeugt dann die *Data Access Objects*, die den Zugriff auf die MySql-Datenbank realisieren und bestimmte Interfaces implementieren. Da die *Business Objects* nur über diese Interfaces auf die DAOs zugreifen, wissen sie nicht welche konkrete Implementierung sie benutzen.

Wenn man eine andere Datenbank oder ein Persistenzframework einbinden möchte, muss man lediglich eine konkrete *Factory*-Klasse sowie die entsprechenden DAOs implementieren und diese *Factory*-Klasse der `DAOFactory` bekannt machen. Für TROIA geschieht dies in der Konfigurationsdatei »troia.properties«. Dort wird der Name der zu verwendenden konkreten *Factory*-Klasse angegeben.

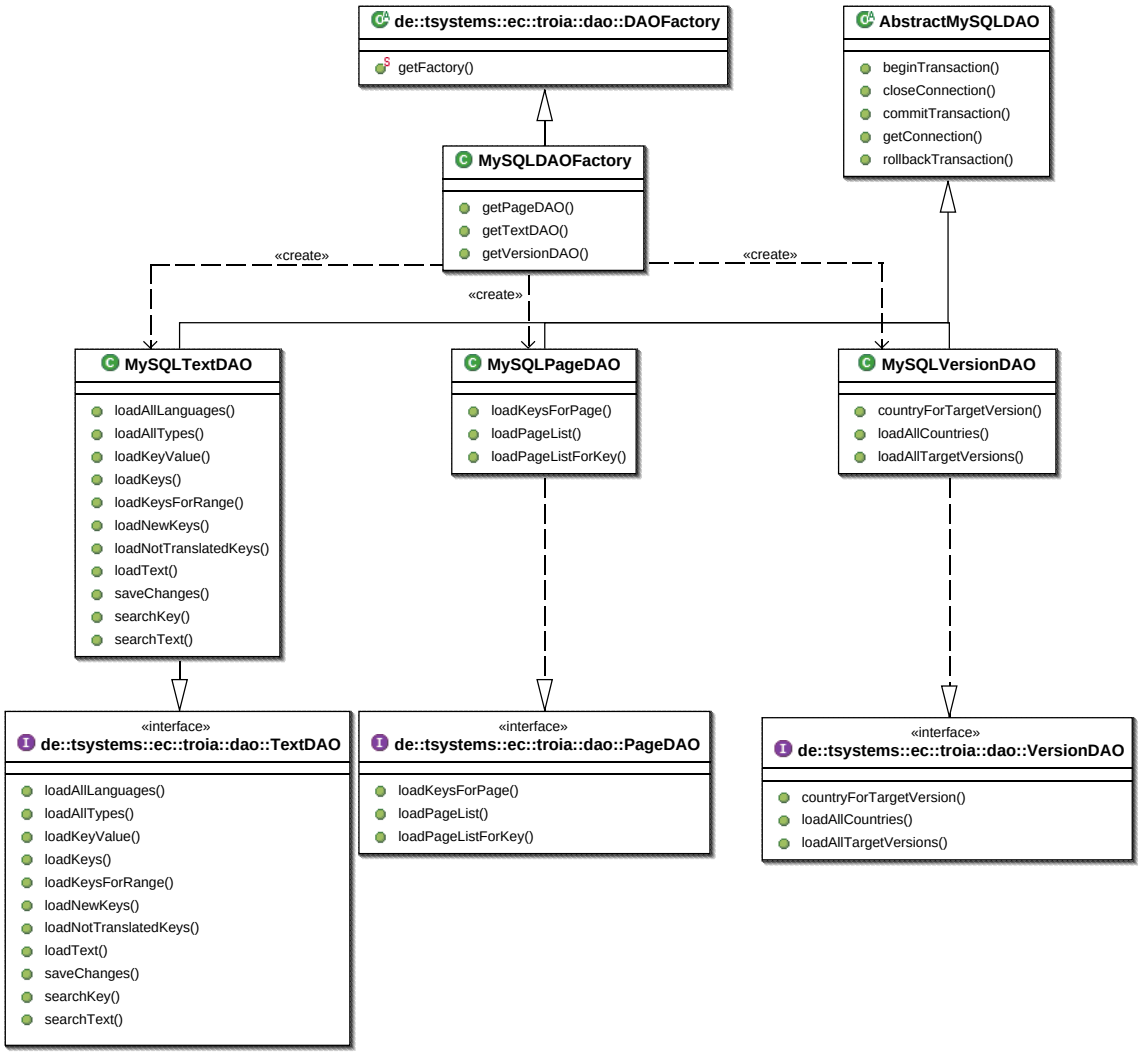

Abbildung 19: UML-Diagramm der Datenzugriffsschicht.

Um auf eine Datenbank zugreifen zu können, muss erst eine Verbindung (engl. connection) aufgebaut werden. Dieser Verbindungsaufbau kann unter Umständen mehrere Sekunden dauern, was die Performance einer Anwendung stark beeinflussen kann. Daher wird in TROIA nicht bei jedem Zugriff auf die Datenbank eine neue *Connection* erzeugt. Die benötigte *Connection* wird aus einem *Connection-Pool* geholt und nach Beendigung der Datenbankoperation wieder an diesen Pool zurückgegeben.

Die *Data Access Objects* benutzen das Interface `javax.sql.DataSource`, um auf die *Connections* in dem Connection Pool zuzugreifen. Als *Connection-Pool* wird der Jakarta Commons DBCP[32] verwendet. Der DBCP implementiert das `DataSource`-Interface und kann über JNDI[33] aufgerufen werden. Über die so erzeugten Connections kann dann auf der Datenbank in die Datenschicht zugegriffen werden.

[32] DBCP - Database Connection Pool . Siehe unter : http://jakarta.apache.org/commons/dbcp/
[33] Siehe auch unter: http://java.sun.com/products/jndi/

8.2.2.5 Die Datenschicht

Die Datenschicht repräsentiert den persistenten Datenspeicher einer Anwendung. In der Regel ist dies eine Datenbank, welche die Daten der Anwendung enthält. Im Fall von TROIA ist dies eine MySql-Datenbank.

8.2.3 Funktionsweise von TROIA

Nachdem in den letzen Kapiteln der technische Aufbau von TROIA beschrieben wurde, wird in diesem Abschnitt die Funktionsweise der Webanwendung TROIA anhand von Screenshots erläutert.

8.2.3.1 IUCCA Login

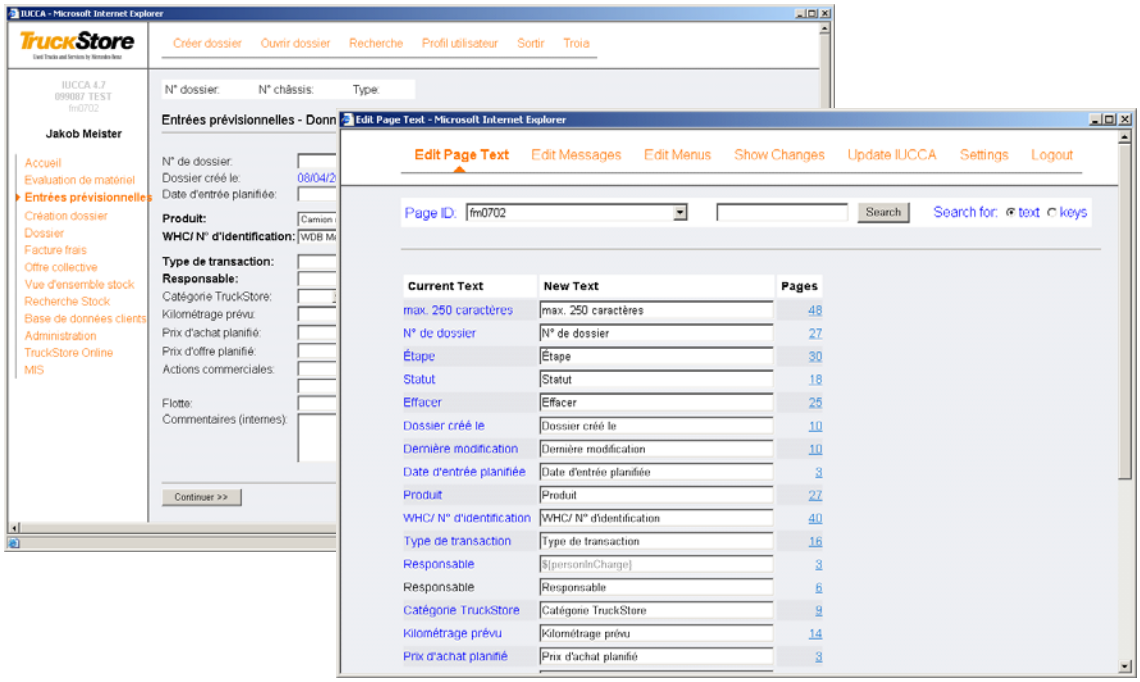

Abbildung 20: Aufruf von TROIA aus IUCCA

Sobald ein IUCCA-Benutzer TROIA über den gleichnamigen Menüpunkt in IUCCA startet, übermittelt das IUCCA-System den Namen des Benutzers, dessen aktuelles *Locale,* die verwendete IUCCA-Version, den Namen der Seite von der aus TROIA aufgerufen wurde und schließlich die URL des verwendeten IUCCA-Servers an TROIA.

Da TROIA keinen Zugriff auf die Benutzerdatenbank von IUCCA hat und man aus Gründen der Benutzerfreundlichkeit keinen separaten TROIA-Login für IUCCA-Anwender haben wollte, führt TROIA keine Authentifizierung der Benutzer durch. Die Rechteverwaltung für den TROIA-Zugriff erfolgt in IUCCA. Nur Benutzer mit dem Recht zum Ändern von Texten haben die Möglichkeit, TROIA aus IUCCA heraus aufzurufen.

Es sind bereits Überlegungen über ein *Single Sign On*-Konzept für IUCCA vorhanden. Allerdings sind sie noch nicht ausreichend definiert, dass TROIA dieses Konzept nut-

zen könnte. Sollte aber in Zukunft diese Anforderung entstehen, müssen lediglich die für den Login zuständigen Action- und Business-Klassen geändert werden, um TROIA in dieses Konzept einzubinden.

Nach dem Aufruf legt TROIA eine Session an, speichert die Daten des Benutzers in die Session und teilt dem Besucher die Rolle eines IUCCA-Benutzers zu. Diese Rolle beschränkt den Anwender dahingehend, dass er nur Textressourcen für das Land und die Version seiner IUCCA-Instanz bearbeiten darf. Anschließend werden die Texte der aktuellen Seite des Benutzers entsprechend seinem *Locale* geladen und angezeigt.

8.2.3.2 Der Menüpunkt »Settings«

Wie man in Abbildung 20 sehen kann, werden dem Benutzer in TROIA die Texte der IUCCA-Seite angezeigt, von der aus er TROIA aufgerufen hat. In der Spalte »Current Text« stehen die aktuellen Texte und in der Spalte »New Text« können diese Texte bearbeitet werden. Über das Drop Down-Menü »Page ID« können die Texte aller weiteren IUCCA-Seiten angezeigt werden. Weiterhin wird dem Benutzter eine Suchfunktion angeboten, über die er entweder nach Texten oder nach deren Schlüssel suchen kann.

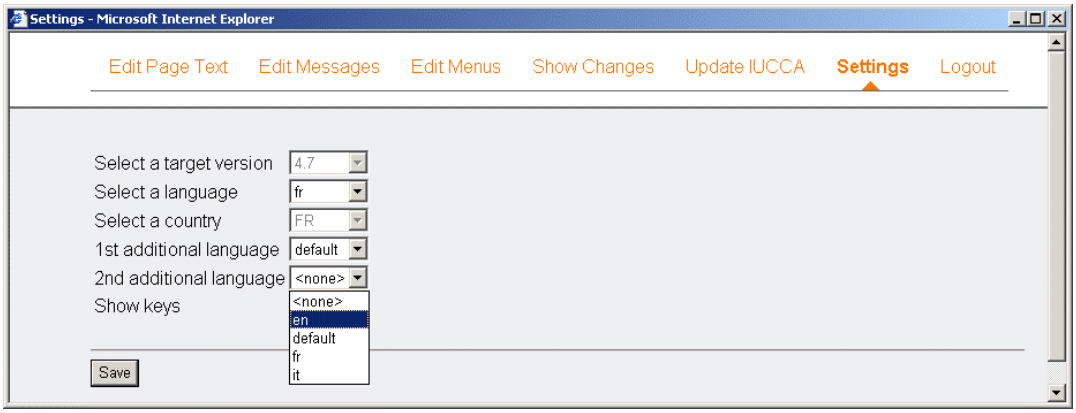

Abbildung 21: Der Menüpunkt »Settings«

Sollte ein Benutzer weitere Informationen zu den in TROIA angezeigten Texten benötigen, so hat er über den Menüpunkt »Settings« die Möglichkeit, sich die Texte in bis zu zwei zusätzlichen Sprachen anzeigen zu lassen. Darüber hinaus können die Schlüsselnamen der Texte dargestellt werden.

Sollte ein Land über mehrere Sprachen verfügen, wie z.B. Belgien französisch und flämisch, so kann in den »Settings« zwischen diesen Sprachen gewechselt werden.

8.2.3.3 Der Menüpunkt »Edit Page Text«

Die Abbildung 22 zeigt die TROIA-Oberfläche, wie sie sich dem Anwender darstellt, sobald er zwei zusätzliche Sprachen und das Anzeigen der Schlüsselnamen ausgewählt hat. Durch die zusätzlich angezeigten Sprachen werden vor allem die Übersetzer unterstützt. Sie können nun prüfen, wie ein bestimmter Begriff in einer anderen Sprache übersetzt wurde. Aber auch für Entwickler ist diese Option hilfreich, da sie schnell die deutsche Übersetzung eines fremdsprachigen Begriffs finden können.

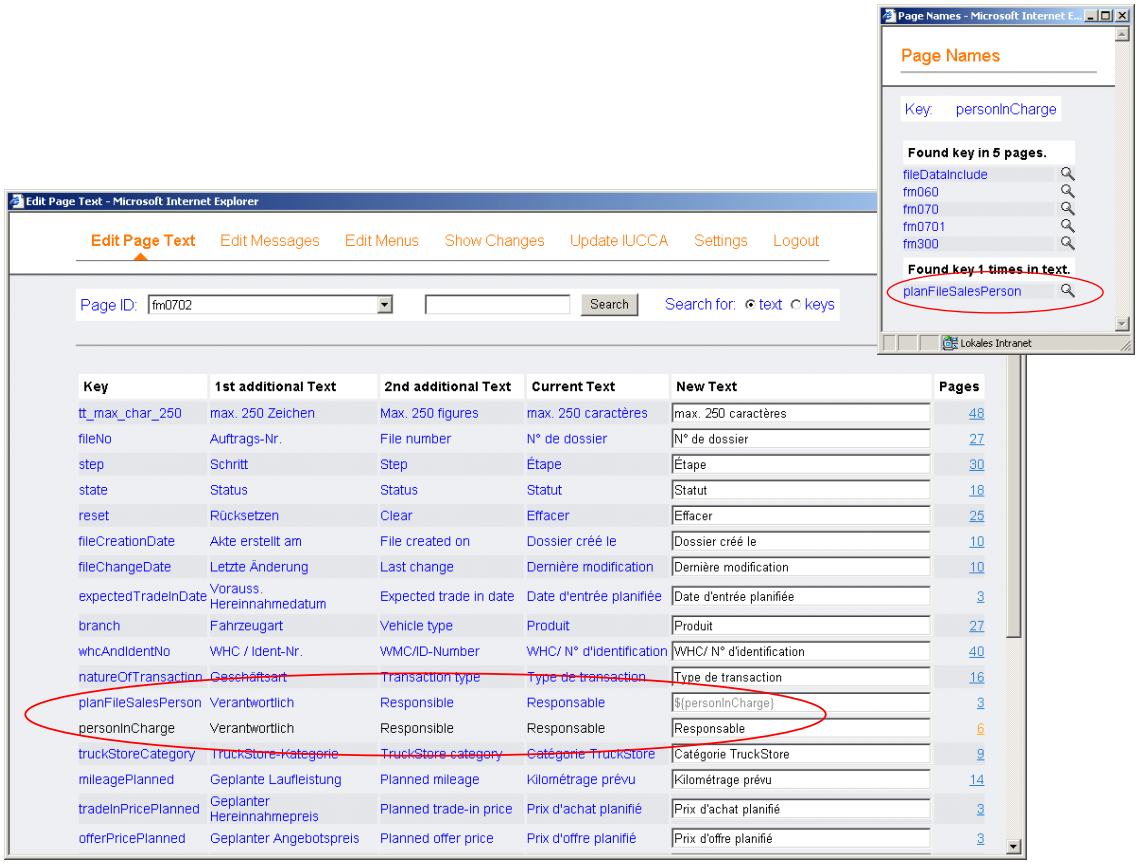

Abbildung 22: Der Menüpunkt »Edit Page Text«

Im alten, Excel-basierten Lokalisierungsprozess stellte das Übersetzen der Schlüssel, die Verweise auf andere Schlüssel beinhalten[34], ein erhebliches Problem dar, da der Übersetzer die Verweise von Hand auflösen musste. Das bedeutet, dass er den Schlüssel, auf den verwiesen wird, in der Excel-Datei suchen musste. Anschließend wurde der Text des Schlüssels übersetzt, ohne dass man ihn im Kontext der anderen Begriffe sah.

TROIA hingegen löst die Verweise auf und stellt die Inhalte der Verweise dar. Die rot markierten Zeilen in Abbildung 22 zeigen solch einen Schlüssel. Wie man sehen kann, enthält der Schlüssel `planFileSalesPerson` in der Spalte »New Text« einen Verweis in der Form `${personinCharge}`. Dieser Verweis referenziert auf den

[34] Siehe Kapitel5.2

Verweis in der Form `${personinCharge}`. Dieser Verweis referenziert auf den Schlüssel `personinCharge`. In der Spalte »Current Text« als auch in den Spalten der zusätzlichen Sprachen von `planFileSalesPerson` wird allerdings nicht der Verweis, sondern der Wert des referenzierten Schlüssels angezeigt.

Soll nun der Text von `planFileSalesPerson` verändert werden, so darf natürlich nicht der Verweis bearbeitet werden, sondern der Text des referenzierten Schlüssels. Deshalb wird in der Zeile unterhalb eines Schlüssels, der einen Verweis enthält, immer der referenzierte Schlüssel angezeigt und farblich von den anderen Schlüsseln abgehoben. Dieser Schlüssel kann nun bearbeitet werden. Nach dem Speichern wird die Änderung in allen Texten, die auf diesen Schlüssel referenzieren, sichtbar.

Generell haben IUCCA-Benutzer nicht das Recht Verweise oder Schlüssel zu bearbeiten. Administratoren hingegen wird dieses Recht zugestanden.

Da Texte in der Regel auf mehreren IUCCA-Seiten verwendet werden und unter Umständen auch als Referenzen in anderen Texten vorkommen, ist es für einen Übersetzer wichtig zu wissen, ob der Text, den er ändern möchte, mehrfach verwendet wird. Wenn dies der Fall ist, sollte der Benutzer wissen, wo und wie der Schlüssel noch verwendet wird. Daher werden in der Spalte »Pages« zu jedem Begriff auch die Anzahl der Seiten bzw. der Verweise angezeigt, in denen dieser Begriff auftritt. Ist die Anzahl größer als eins, so wird der Begriff noch in einem anderen Kontext verwendet und es wäre sinnvoll vor dem Ändern dieses Begriffs zu prüfen, welche Auswirkungen die Änderung in dem anderen Kontext hat.

Um sich die Details über die Verwendung anzeigen zu lassen, kann der Benutzer auf die Anzahl in der Spalte »Pages« klicken. Diese Aktion öffnet ein *Popup*-Fenster, wie es in Abbildung 22 dargestellt wird. Dieses Fenster zeigt sowohl die Namen der einzelnen JSPs, als auch die Schlüsselnamen in denen der Begriff verwendet wird. Über das Lupensymbol werden im TROIA-Hauptfenster die Texte der entsprechenden Seite bzw. der referenzierenden Schlüssel geladen und der Anwender kann beurteilen, welche Auswirkungen seine Änderungen haben könnten.

8.2.3.4 Der Menüpunkt »Edit Messages«

Für das Bearbeiten von Meldungen, die das IUCCA-System an Anwender ausgibt, steht unter dem Menüpunkt »Edit Messages« eine entsprechende Oberfläche zur Verfügung. In IUCCA sind diese Meldungen keiner bestimmten Seite zugeordnet, sondern werden nach Bedarf dynamisch geladen.

Der Benutzer kann sich die Meldungen entweder über ein Drop Down-Menü in alphabetisch sortierten Blöcken anzeigen lassen, oder über die Suchfunktion nach einzelnen Meldungen bzw. deren Schüssel suchen. Die Einteilung der Meldungen in alphabetische Blöcke ist vor allem als Bearbeitungshilfe beim erstmaligen Übersetzen der Sprachressourcen gedacht. Benötigt man eine bestimmte Meldung, so ist die Suche über die Suchfunktion wesentlich effektiver.

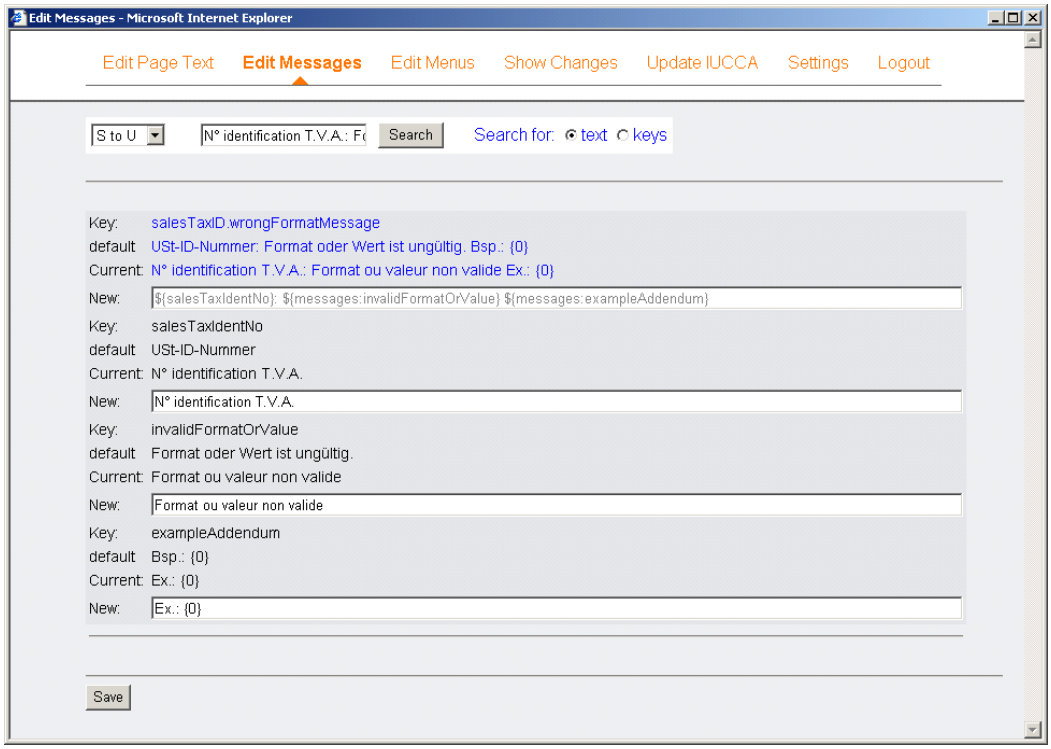

Abbildung 23: Der Menüpunkt »Edit Messages«

In Abbildung 23 wurde eine Nachricht mit Hilfe der Suchfunktion gefunden. Man kann erkennen, dass diese Nachricht aus drei Verweisen besteht, von denen der erste auf einen Schlüssel in den »text.properties« verweist und die anderen beiden auf Schlüssel in den »messages.properties«. Spätestens hier sollte klar sein, dass das Übersetzen, im Speziellen von dieser Art von Schlüsseln, bisher keine triviale Aufgabe war. Um die zu übersetzende Nachricht zu ermitteln, musste der Anwender drei Schlüssel aus zwei unterschiedlichen Dateien zusammentragen, bevor er mit der Bearbeitung dieser Nachricht beginnen konnte

8.2.3.5 Der Menüpunkt »Edit Menus«

Über den Menüpunkt »Edit Menus« gelangt man auf die Oberfläche, in der die Menüs von IUCCA editiert werden können. Diese Oberfläche ist im Prinzip identisch mit der aus dem Menüpunkt »Edit Page Text«, außer dass auch hier eine Darstellung in alphabethisch sortierten Blöcken möglich ist, da auch die Menüs nicht an bestimmte Seiten gebunden sind. Alle Menütexte aus den »menus.properties« verweisen auf Schlüssel in den »text.properties«.

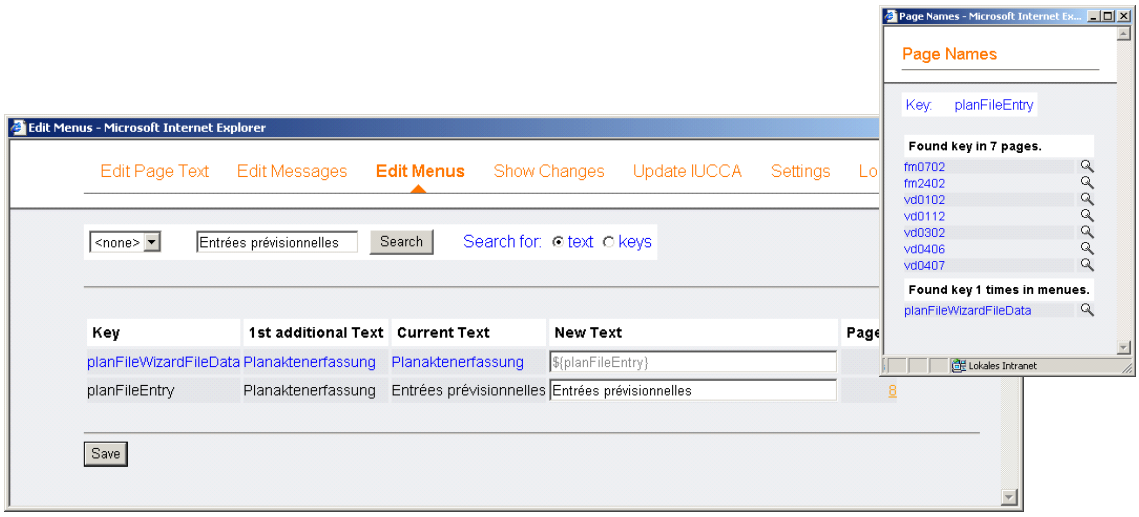

Abbildung 24: Der Menüpunkt »Edit Menus«

In Abbildung 20 sieht man, dass der Menütext »Entrées prévisionnelles« zu lange für das aktuelle Layout der Seite ist. Ein Übersetzer könnte jetzt, wie in Abbildung 24 gezeigt, nach diesem Text suchen und ihn abkürzen. Dabei muss er allerdings beachten, dass dieser Begriff außer im den Menüs auch auf sieben weiteren JSPs verwendet wird.

8.2.3.6 Der Menüpunkt »Show Changes«

Nachdem eine neue Version gebaut wurde, ist es die Aufgabe des Übersetzers, die neu hinzugekommenen Schlüssel zu lokalisieren. Damit der Übersetzer aber nicht jede Seite und alle Nachrichten nach neuen Schlüsseln durchsuchen muss, kann er sich unter dem Menüpunkt »Show Changes« alle neu hinzugekommenen Texte anzeigen lassen.

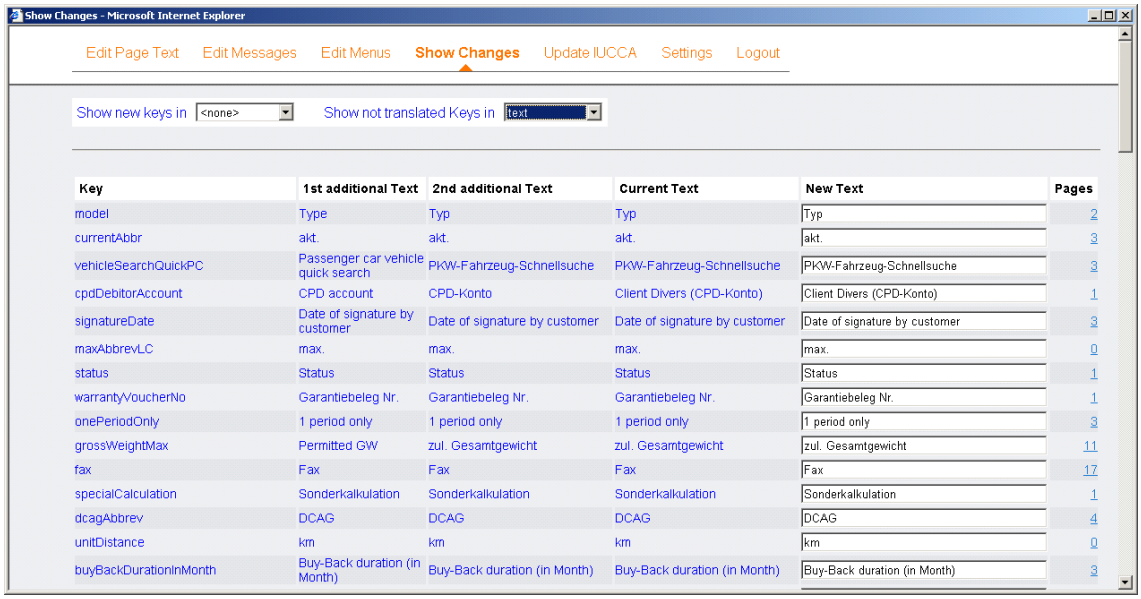

Abbildung 25: Der Menüpunkt »Show Changes«

Über das *Drop Down*-Menü »Show new keys« können die Texte ermittelt werden, welche seit der letzten Version neu eingefügt wurden. Durch das zweite *Drop Down*-Menü hat der Anwender die Möglichkeit alle Texte zu finden, die bisher noch nicht übersetzt worden sind. Dies beinhaltet sowohl die neuen Texte als auch Texte aus älteren Versionen.

8.2.3.7 Der Menüpunkt »Update IUCCA«

Nachdem der Anwender Änderungen in den Textressourcen vorgenommen hat, kann er über den Menüpunkt »Update IUCCA« die Aktualisierung von IUCCA bewirken. Dadurch ist er in der Lage, sofort zu prüfen, ob seine Änderung wirklich in den Kontext der aktuellen Seite passt und welche Auswirkungen sie auf das Layout der Seite hat. Außerdem können dadurch Fehler unmittelbar im aktuellen System korrigiert werden, ohne dass sich der Anwender an einen Entwickler wenden muss.

Für die Aktualisierung von IUCCA liest TROIA die dem *Locale* des Anwenders entsprechenden Sprachressourcen aus der Datenbank, wandelt dann alle Zeichen die außerhalb des ISO 8859-1 Schriftsatzes liegen in Java *Unicode-Escapes* um und sendet schließlich die neu erzeugten Properties-Dateien über einen HTTP-POST-Request

an die entsprechende IUCCA-Schnittstelle. Diese Schnittstelle wir durch ein Servlet realisiert, das den *Request* annimmt und veranlasst, dass die Properties-Dateien in das entsprechende Verzeichnis geschrieben werden und das `ResourceBundle` die neuen Ressourcen lädt. Anschließend kann der Benutzer seine Änderungen an der Oberfläche sehen.

8.2.3.8 Administrator

Anwender mit der Rolle »Administrator«, haben direkten Zugang zu TROIA. Diese Rolle erlaubt dem Anwender die Sprachressourcen für alle Länder und Versionen zu bearbeiten. Nach dem Login wird der Administrator auf eine Seite geleitet, auf der er die gewünschte Version, Sprache und das Land auswählen kann. Diese Einstellungen kann er jederzeit in dem Menüpunkt »Settings« verändern. Anschließend kann der Administrator die Sprachressourcen bearbeiten. Im Unterschied zu IUCCA-Benutzern wird ihm jedoch der Menüpunkt »Create Properties« angezeigt.

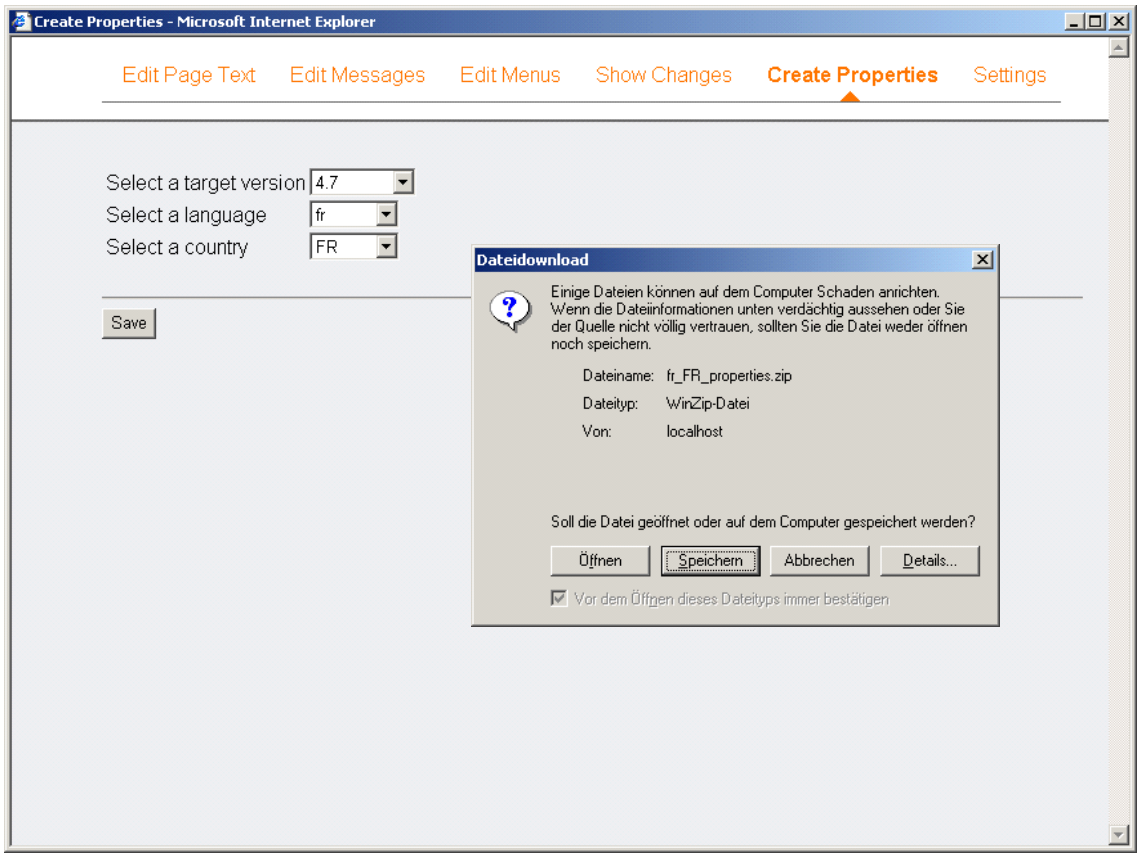

Abbildung 26: Der Menüpunkt »Create Properties«

Unter dem Menüpunkt »Create Properties« kann der Administrator Properties-Dateien für alle in TROIA gespeicherten Sprachressourcen erzeugen. Nach dem er eine Version und ein *Locale* ausgewählt hat, werden alle Sprachressourcen für diese Parameter in ein Zip-Archiv gepackt und zum Download angeboten. Nach dem Speichern und

Entpacken des Archivs, können die Properties-Dateien in das CVS eingepflegt werden und stehen dann zum Bauen einer neuen IUCCA-Instanz zur Verfügung.

9 Ausblick

Die Internationalisierung gewinnt in der Softwareentwicklung immer mehr an Bedeutung. Insbesondere Softwarelösungen, die für international agierende Konzerne entworfen werden, können sich dieser Anforderung nicht mehr entziehen.

Für die Internationalisierung von Software bieten moderne Programmiersprachen, wie Java, viele hilfreiche Funktionalitäten an. Beim Entwurf einer Anwendung muss aber auch geprüft werden, ob die verwendeten *Back-End* Systeme den Anforderungen der Internationalisierung genügen. So haben viele ältere Datenbanksysteme Probleme bei der Unterstützung von Unicode.

Ein weiterer Aspekt, der nicht unterschätzt werden darf, ist die Lokalisierung einer Anwendung. Wie in dieser Arbeit gezeigt wurde, kann der Lokalisierungsprozess unter den entsprechenden Umständen sehr zeitaufwändig und fehleranfällig werden.

Durch die im Rahmen dieser Diplomarbeit entstandene Anwendung TROIA wurden viele Schritte im Lokalisierungsprozess für IUCCA automatisiert. Allerdings besteht hier noch Potential für weitere Automatisierungsschritte. So wäre eine direkte Anbindung von TROIA an das CVS denkbar. Dadurch könnten die Entwickler die Properties-Dateien direkt aus TROIA ins CVS einpflegen.

Weiterhin könnte der Funktionsumfang von TROIA auf die Druckvorlagen von IUCCA ausgedehnt werden. Diese Vorlagen werden zum Erstellen von Rechnungen, Preisschildern und sonstigen für IUCCA relevanten Ausdrucken verwendet. Da sie als XSLT-Stylesheets vorliegen, sind sie momentan noch nicht über TROIA lokalisierbar.

Zum Abschluss möchte ich noch an einem Beispiel aufzeigen, wie wichtig es sein kann, auch solche Software zu internationalisieren, für die eigentlich keine Lokalisierung geplant ist:

Um nicht auch einen Lokalisierungsprozess für TROIA aufsetzen zu müssen, sollten die TROIA-Oberflächen für alle Länder in Englisch dargestellt werden. Dennoch wurde TROIA internationalisiert aber nicht lokalisiert. Bei der Einführung von TROIA in Italien stellte sich dann heraus, dass »troia« ein italienisches Schimpfwort ist. Durch die Lokalisierung des Begriffs »troia« für italienische Anwender konnte dieses Problem schnell behoben werden.

Literaturverzeichnis

[AlCrMa01]
Alur,D.; Crupi,J.; Malks, D. (2001): Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall / Sun Microsystems Press

[Amshoff04]
Amshoff, C. (2004): Internationalisierung von Webanwendungen.
Java Magazin, 11.

[Bodoff02]
Bodoff, S. (2002) : Java Servlet Technologie.
http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html [01.04.2005]

[DAVIS05]
Davis, M. (2005): Security Considerations for the Implementation of Unicode and Related Technology. http://www.unicode.org/reports/tr36/tr36-2.html [18.03.2005]

[Garnier03]
Garnier, J.M. (2003): Struts 1.1 Controller UML diagrams.
http://rollerjm.free.fr/pro/Struts11.html [05.04.2005]

[Green05]
Green, D. (2005) : The Java Tutorial. Trail: Internationalization.
http://java.sun.com/docs/books/tutorial/i18n/intro/index.html [14.März 2005]

[GoF04]
Gamma, E.; Helm, R., Johnson, R.; Vlissides, J. (2004): Entwurfsmuster.
Addison-Wesley, München.

[Gupta05]
Gupta, S. (2005): The Mysteries of Business Object.
http://javaboutique.internet.com/tutorials/businessObject/ [09.04.2005]

[Leutner05]
Leutner, M (2005): Projektübersicht. Foliensatz.
Unveröffentlicht. T-System International GmbH.

[Krämer04]
Krämer, H (2004): Entwurf einer effizienten Ressourcenadministration für mehrländerfähige Systeme. Diplomarbeit. Bibliothek der Hochschule der Medien.

[TuBe03]
Tuner, J ; Bedell, K. (2003): Struts. JSP-Applikationen mit Jakarta Struts, JBoss und Apache Axis. Addision-Wesley.

[Seshadri99]
Seshadri, G. (1999): Understanding JavaServer Pages Model 2 architecture.
http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html. [03.04.2005]

[SiStJo02]
Singh, I.; Stearns, B.; Johnson, M.; Enterprise Team (2002): Designing Enterprise Applications with the J2EE Platform. Addison-Wesley.

[Shirah02]
Shirah, J. S. (2002): Java internationalization basics.
http://www-128.ibm.com/developerworks/edu/j-dw-javai18n-i.html [14.03.2005]

[Struts05]
Apache Struts (2005): User Guide.
http://struts.apache.org/userGuide/index.html [06.04.2005]

[Sun05]
Sun Microsystems (2005): Products & Technologies: Java Technology.
http://java.sun.com [14.03.2005]

[LiOk04]
Lindenberg, N und Okutsu, M (2004): Supplementary Characters in the Java Platform.
http://java.sun.com/developer/technicalArticles/Intl/Supplementary/ [22.03.2005]

[UML05]
OMG -Object Management Group (2005) : UML Resource Page.
http://www.uml.org [01.04 .2005]

[Unicode05]
Unicode (2005): Unicode 4.0.0
http://www.unicode.org/versions/Unicode4.0.0 [16.03.2005]

[WikiUTF05]
Wikipedia (2005):UTF-8
http://de.wikipedia.org/wiki/UTF-8.[14.04.2005]

[WikiMySQL05]
Wikipedia (2005):MySQL.
http://de.wikipedia.org/wiki/Mysql[14.04.2005]

[WikiWF05]
Wikipedia (2005): Wasserfallmodell.
http://de.wikipedia.org/wiki/Wasserfallmodell [29.03.2005]